Titre: Title:	Conception et implémentation de processeurs dédiés pour des systèmes de traitement vidéo temps réel
Auteur: Author:	Gérard Armand Bouyela Ngoyi
Date:	2009
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bouyela Ngoyi, G. A. (2009). Conception et implémentation de processeurs dédiés pour des systèmes de traitement vidéo temps réel [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/121/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/121/
Directeurs de recherche: Advisors:	Yvon Savaria, & Pierre Langlois
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

CONCEPTION ET IMPLÉMENTATION DE PROCESSEURS DÉDIÉS POUR DES SYSTÈMES DE TRAITEMENT VIDÉO TEMPS RÉEL

GÉRARD ARMAND BOUYELA NGOYI

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU

DIPLÔME DE MAITRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

MARS 2009

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION ET IMPLÉMENTATION DE PROCESSEURS DÉDIÉS POUR DES SYSTÈMES DE TRAITEMENT VIDÉO TEMPS RÉEL

présenté par : BOUYELA NGOYI Gérard Armand

en vue de l'obtention du diplôme de : Maitrise ès sciences appliquées

a été dûment accepté par le jury constitué de :

M. BOYER François-Raymond, Ph. D., président;

M. LANGLOIS Pierre J.M, Ph. D., membre et directeur de recherche;

M. SAVARIA Yvon, Ph. D., membre et codirecteur de recherche;

M. BOIS Guy, Ph. D., membre;

DÉDICACE

À mes parents et à mes frères et sœurs.

REMERCIEMENTS

Je tiens à exprimer ma reconnaissance envers mon directeur de recherche, le Professeur Pierre Langlois, Ph. D, pour l'opportunité qu'il m'a offert de travailler sur un sujet aussi passionnant. Son assistance, sa perpétuelle motivation et son sens prononcé de l'organisation m'ont été très précieux pour l'aboutissement de mes recherches.

J'adresse également ma gratitude à mon co-directeur, le Professeur Yvon Savaria, Ing. Ph. D. Son expertise dans le domaine des systèmes embarqués et VLSI a aussi été un point important dans la réussite de nos travaux.

J'aimerais aussi remercier les membres du jury qui ont accepté d'évaluer le travail réalisé ainsi que le Département de Génie Informatique et Génie Logiciel de l'École Polytechnique de Montréal pour m'avoir fourni des équipements nécessaires à la réalisation de ce projet de même qu'un cadre de travail agréable et stimulant.

Je souligne la grande générosité et les efforts de mes amis Kéfil Landou, Roméo Honvo, Camille Hombouhiry, Thierry Saint-Victor, Patrick Divoungui, Stéphane Tchoulack, et Nadège Ditsambou pour leur aide et leur soutien moral.

RÉSUMÉ

Les systèmes de traitement vidéo se caractérisent par des demandes de performance de plus en plus exigeantes. Les nouvelles normes, telles le HMDI 1.3 (High Definition Media Interface), requièrent des bandes passantes allant jusqu'à 340 Méga-pixels par seconde et par canal. Il en découle que les processeurs traitant ce type d'information doivent être très performants. Les nouvelles méthodologies de conception basées sur un langage de description d'architecture (ADL) apparaissent pour répondre à ces défis. Elles nous permettent de concevoir des processeurs dédiés de bout en bout, avec un maximum de flexibilité.

Cette flexibilité, grande force de ce type de langage (tels LISA 2.0), nous permet par ajout d'instructions spécialisées et modification de l'architecture (ajout de registres spécialisés, modification de largeur de bus), de créer un processeur dédié à partir d'architectures de base considérées comme des processeurs d'usage général.

Dans le cadre de nos travaux, nous nous sommes concentrés sur un type d'algorithmes de traitement d'image, le désentrelacement. Le désentrelacement est un traitement qui permet de reconstruire une séquence vidéo complète à partir d'une séquence vidéo entrelacée pour des raisons telles que la réduction de bande passante. Tout au long de nos travaux, nous avons eu un souci constant de développer des méthodologies, les plus générales possibles, pouvant être utilisées pour d'autres algorithmes.

L'une des contributions de ce mémoire est le développement d'architectures de test complètes et modulaires, permettant d'implémenter un processeur de traitement vidéo

temps réel. Nous avons également développé une interface de gestion de RAM qui permet au cours du développement des processeurs de les tester sans modifier le système au complet.

Le développement de deux méthodologies innovatrices représente un apport supplémentaire dans la conception de processeurs dédiés. Ces deux méthodologies, qui se basent sur un langage ADL, sont synergiques et permettent d'implémenter et d'accélérer des algorithmes de traitements vidéo temps réel. Nous obtenons dans un premier temps un facteur d'accélération de 11 pour la première méthodologie puis un facteur d'accélération de 282 pour la deuxième.

ABSTRACT

Video processing systems are characterized by rising performance specifications. New standards such as the HDMI 1.3 require bandwidths as high as 340 megapixels per second and per channel, resulting in greater information processing power. New conceptual methodologies based on architectural descriptions (ADL) seem to respond to this challenge. Design methods and languages for architectural descriptions (such as LISA 2.0), allow developing tailor-made high performance processors in a very flexible way.

The flexibility of these languages let the user add specialized instructions to an instruction set processor. They also allow modifying its architecture to create a processor with much improved performance compared to some baseline general purpose processsor.

Our study focuses on a specific type of video processing algorithm called deinterlacing. Deinterlacing allows reconstructing a complete video sequence from an interlaced video sequence. Despite this algorithmic focus, in the course of this study, we were concerned with developing broadly applicable methodologies usable for other algorithms.

This thesis aims to contribute to the existing body of work in the field by developing complete and modular test architectures allowing to implement processors capable of real time video processing. The development of two innovative design methodologies represents an additional contribution. These synergetic methodologies are based on ADL (Architecture Description Language). Our results confirm that they allow

implementing processors capable of real-time video processing. We obtained an acceleration factor of 11 with a first design method and the acceleration factor was further improved to 282 with a second method.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ANNEXES

LISTE DES ACRONYMES

ADL	Architecture Description Language
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Integrated Processor
DSP	Digital Signal Processor
DVI	Digital Visual Interface
ELA	Edge-based Line Average
FFT	Fast Fourier Transform
FIFO	First In, first Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
HDMI	High Definition Multimedia Interface
HDTV	High Definition Television
ISD	Instruction Set Designer
JTAG	Joint Test Action Group
NTSC	National Television System Committee
RISC	Reduced Instruction Set Computer
RGB	Red Green Blue
SIMD	Single Instruction Multiple Data
TIE	Tensilica Instruction Extension
VGA	Video Graphics Array

VHDL	VHSIC (Very High Speed Integrated Circuits) Hardware Description Language
VLIW	Very Long Instruction Word
YCrCb	Luminance (Y), Différence de Chrominance rouge (Cr), Différence de Chrominance bleu (Cb)

INTRODUCTION

L'industrie des produits multimédias fonctionne aujourd'hui à un rythme effréné. Ce phénomène engendre une recherche constante d'amélioration de la qualité vidéo aboutissant à l'apparition de nouvelles normes et à la disparition d'autres. Par exemple, on remarque en scrutant le marché actuel, que déjà certains nouveaux ordinateurs ne supportent plus le VGA (Vidéo Graphics Array) laissant la place aux normes HDMI (High Definition Multimedia Interface) et DVI (Digital Visual Interface) qui s'imposent de plus en plus. Dans le même ordre d'idée, les écrans de télévision à tubes cathodique ont fait place aux téléviseurs à haute définition.

Cette recherche croissante de qualité d'image a bien entendu un prix. Elle se traduit par la demande d'algorithmes de plus en plus complexes et, par ricochet, de matériel ayant des niveaux de performance élevés pour respecter les contraintes imposées par ces nouvelles normes.

En effet, en effectuant une analyse de l'évolution des processeurs, on remarque que nous sommes bien loin du premier processeur d'Intel qui a vu le jour en 1971. Ce processeur, baptisé 4004, fonctionnait avec des registres de 4 bits.et était composé de 2300 transistors [1,2]. De nos jours nous avons sur le marché des processeurs à double cœur qui fonctionnent à des fréquences de 3.2 GHz, tandis que le nombre de transistors atteint 42 million [3]. Conjugué à ce phénomène d'explosion du matériel, on retrouve l'aspect crucial d'être le premier sur le marché, ce qui implique des contraintes drastiques sur le temps de développement, de test et de production.

En résumé, on veut produire du matériel de plus en plus performant du point de vue de la vitesse de calcul, de l'espace occupé et de la puissance consommée, tout en réduisant le temps de conception. De cette problématique découle l'apparition de nouvelles méthodes de conception de systèmes de traitement d'information qui prennent en compte la technologie actuelle de développement matérielle ainsi que des attentes des consommateurs.

Objectifs du projet

Ce travail s'inscrit dans le développement de processeurs dédiés pour le traitement vidéo temps réel. Nous nous attarderons tout particulièrement sur des algorithmes de désentrelacement destinés aux systèmes de télévision haute définition (HDTV).

Plus spécifiquement, les objectifs suivants sont poursuivis pour ce programme de recherche :

1. Implémenter un processeur vidéo pour réaliser des algorithmes de désentrelacement vidéo en temps réel pour différents formats d'image comme les normes NTSC et HDTV.
2. Concevoir un environnement de test complet, permettant de visualiser les résultats de l'implémentation matérielle.
3. Définir une méthodologie de conception basée sur une description logicielle des algorithmes exploitant les architectures matérielles développées.
4. Définir une méthodologie de conception basée sur un ADL (Architecture Description Language) permettant l'accélération d'algorithmes de traitement vidéo.

Organisation du mémoire

Ce mémoire contient quatre chapitres. Le chapitre 1 fait un survol de la théorie sous jacente à la conception de processeurs dédiés, des techniques de désentrelacement ainsi qu'à la présentation de notre environnement de travail.

Dans le chapitre 2, nous passons en revue la plateforme de développement et l'exploration architecturale qui en découle, ainsi que les méthodologies proposées et développées, pour la réalisation des processeurs dédiés avec un ADL (Architecture Description Language).

Dans le chapitre 3, nous présentons les résultats des différentes méthodologies et de l'implémentation de notre système.

Dans le chapitre 4, nous présentons les résultats de l'implantation matérielle de l'environnement de développement et de test des différentes méthodologies et de l'implémentation de notre système.

Dans le chapitre 5, nous effectuons une analyse des différents résultats obtenus et présentons les résultats de synthèse.

Enfin, en conclusion, nous rappelons les objectifs atteints et donnons des pistes pour de futures recherches.

CHAPITRE 1. REVUE DE LITTÉRATURE ET THÉORIE

Ce chapitre est scindé en trois parties. Dans un premier temps, nous présentons un survol des différentes catégories de processeurs et particulièrement des méthodes de conception de processeurs dédiés. Nous nous attardons aussi sur les défis rencontrés lors de la conception de processeurs dédiés pour le traitement vidéo temps réel.

Nous présentons, par la suite, en détail l'environnement de travail et le langage LISA utilisés comme descripteur de processeurs. Pour finir, nous présentons les concepts d'entrelacement et de désentrelacement et différents algorithmes s'y rattachant.

1.1. Conception de processeurs dédiés

1.1.1. Les différentes classes de processeurs

La conception de processeurs se fait à l'aide de trois types d'architectures :

- Les processeurs d'usage général : ce sont des processeurs permettant d'exécuter différents types d'application; pour un niveau de performance donné, ils sont en général coûteux mais très flexibles, car ils peuvent exécuter plusieurs types d'algorithmes.
- Les ASIC (Application Specific Integrated Circuits): il s'agit de circuits intégrés spécialisés pour une application spécifique. Ils ont pour avantage d'être optimisés pour un type d'application au détriment du coût et de la flexibilité. En

effet une fois conçus, il est en général impossible de les modifier car leur architecture interne est souvent cablée et n'est ni programmable ni configurable.

- Les ASIP (Application Specific Integrated Processors) qui sont des circuits logiques dont le jeu d'instructions est adapté à une application spécifique. Cette spécialisation offre un compromis entre la flexibilité d'un processeur à usage général et les performances d'un ASIC.

Les ASIP utilisant comme technologie d'implantation un FPGA, sont des candidats idéaux pour créer des prototypes de processeurs car ils nous offrent la flexibilité requise à l'exploration architecturale ainsi que des performances acceptables et suffisantes pour le traitement d'image en temps réel. Ils permettent aussi, au même titre que les ASICs et certains processeurs à usage général, d'exploiter du parallélisme, ce qui facilite le traitement de grands volumes de données et d'opérations complexes à une vitesse relativement élevée.

On retrouve ainsi plusieurs alternatives pour concevoir des processeurs dédiés. Tout d'abord, il est possible de concevoir des processeurs sur mesure qui se caractérisent par des architectures très spécialisées et qui requièrent des performances extrêmes (grande bande passante, consommation de puissance optimisée) [6, 33]. C'est dans cet ordre d'idée qu'on retrouve ces processeurs dans la famille ASIC, où l'optimisation et le design se concentre jusqu'au niveau transistor ou couche (dessin de masques). Cette conception de bout-en-bout s'avère parfois nécessaire pour atteindre des hauts niveaux de performance, au détriment du temps de développement et du coût qui se voient augmentés.

L'alternative utilisée le plus souvent est de se baser sur un canevas, où l'on retrouve déjà un processeur de base avec un jeu d'instructions préexistantes. À partir de ce processeur, on étend le jeu d'instructions en y ajoutant des instructions spécialisées et/ou registres spécialisés.

À ce niveau aussi on a plusieurs catégories. On retrouve par exemple des DSP, dont l'architecture est optimisée pour effectuer des calculs complexes en un coup d'horloge, mais aussi pour accéder très facilement à un grand nombre d'entrées-sorties [7, 8]. Dans ces processeurs, il y a souvent plusieurs modules de calcul spécifiques aux traitements (FFT, FIR, Multiplieur/diviseur à virgule flottante) [9].

On retrouve aussi des processeurs configurables tels Xtensa [10] qui proposent un jeu d'instruction de base avec la possibilité d'ajouter des TIE [11] (Tensilica Instruction Extension). L'idée est la même, partir d'un processeur d'usage général et y ajouter des instructions spécialisées.

Néanmoins, même avec cette méthodologie robuste, on retrouve des limites telles la largeur du bus fixe ou encore le jeu d'instructions de base qui ne peut-être supprimé (overhead). Pour palier à ces limites, de nouvelles méthodologies qui reposent sur un langage de description d'architecture (ADL) [4] commencent à apparaître et à s'imposer sur le marché.

Pour finir, on retrouve des systèmes introduisant la notion de « co-processing » [12, 13,43]. L'idée ici est de faire prévaloir le vieil adage : « diviser pour mieux régner ». Dans ce cadre, plusieurs processeurs s'occupent d'exécuter différentes parties du traitement global. Typiquement on retrouve un processeur spécialisé s'occupant

d'effectuer des calculs critiques et complexes et un processeur d'usage général permettant le contrôle du système.

Dans ce projet de maîtrise, nous implémentons un système de traitements temps réel qui serait difficilement réalisable à l'aide d'un processeur d'usage général de base, à cause de l'accès séquentiel qu'il a aux données.

1.1.2. Les défis rencontrés en traitement vidéo.

Tout d'abord nous avons plusieurs types de traitements en imagerie. En catégorisant ces différents types nous retrouvons [4] :

- Le mode « flot de données » : Dans ce mode, nous avons des débits de données égaux en entrée et en sortie.
- Le mode «hors ligne» : Les données sont contenues dans une mémoire et il n'y a pas de contrainte de temps, l'exécution du processus est restreinte par le temps d'accès en mémoire.
- Le mode «hybride» : C'est un mélange des deux modes. Il consiste à une mise en tampon des données d'entrées dans une mémoire pour de futurs traitements. Dans ce cas l'ajout de ces tampons introduit des temps de latence pour sortir les premières données valides. Par la suite nous avons un débit de sortie des données constant qui peut être supérieur ou égal au débit d'entrée.

Dépendamment de ces types de traitements, les concepteurs de processeurs sont confrontés a plusieurs contraintes qui se résument comme suit :

- Temps : Il faut effectuer tous les traitements dans le temps imposé par le débit des données. Si l'on considère une séquence vidéo avec 60 images par seconde, nous avons une contrainte de temps pour traiter chaque image de 16,67 ms/image. En considérant une image couleur RGB de 2,25 Mo (1024x768, 1 octet par couleur), et que l'on effectue 30 opérations par octet, nous obtenons une fréquence minimale de notre système.

$$T\max \approx \frac{16{,}67(s/image)}{1024\times 768\times 3(octet/image)\times 30(operation/octet)} \approx 230(ps/operation)$$

D'où

$$F\min \approx 4{,}34GHz \text{ , avec 1 opération par cycle}$$

Cette valeur nous donne une idée des efforts à fournir au niveau de l'implémentation quand on se rend compte que 30 opérations par octet est une valeur que l'on peut aisément multiplier par 10 pour des algorithmes de traitement vidéo plus complexes. [5]

- Ressources utilisées : Une seule image RGB de 1024 × 768 pixels requiert 2.25 Mo de mémoire. Dépendamment des algorithmes de traitements, on peut vouloir conserver plusieurs images, imposant ainsi une contrainte supplémentaire pour le développement du processeur.
- Bande passante : Il est important de fournir des efforts de conception des accès en mémoire en optimisant l'écriture et la lecture des données. Un accès lent aux données externes au processeur réduit considérablement ses performances. Même si on fournit des efforts importants d'optimisation des traitements

effectués par un processeur, cette optimisation devient inutile si l'accès aux données que l'on doit traiter est lent.

1.1.3. Langages de description d'architecture (ADL)

De plus en plus, de nouvelles méthodes permettent de décrire l'architecture de notre processeur de bout en bout mais de façon assistée. Les langages de description d'architecture s'inscrivent dans cette optique de méthodologie dites ADL. Dépendamment de leur aptitude à décrire un processeur, les langages ADL sont classifiés [34] en trois catégories : comportemental, structurel et mixte. L'ADL comportemental permet uniquement de modifier le jeu d'instructions des processeurs développés à l'aide de ces méthodologies. On retrouve dans ce cadre les langages tels ISPS [35], nML [36], ou encore ISDL [37].

Les ADL structurels quant à eux permettent de décrire uniquement l'architecture d'un processeur. On retrouve dans cette catégorie les langages MIMOLA [38] et UDL/I [39].

Pour finir, les ADL mixtes permettent de couvrir le jeu d'instructions ainsi que la description matérielle du processeur. On retrouve dans cette catégorie les langages tels LISA [40], EXPRESSION [41] ou encore MADL [42].

Dans [15] on retrouve plusieurs types de langages ADL et les outils supportés par ces derniers. Une comparaison que l'on retrouve dans cet article est présentée au tableau 1.1.

Tableau 1.1 : Comparaison entre différents ADL, X : supporté O : supportés avec des restrictions

	MIMO	UDL/I	nML	ISDL	HMDES	EXPRES-SION	LISA	RADL	AIDL
Génération du compilateur	X	X	X	X	X	X	X		
Génération du simulateur	X	X	X	X	X	X	X	X	X
Simulation ‘‘Cycle-accurate’’	X	O		X	X	X	X	X	O
Vérification formelle						O			O
Génération de matériels (HDL,Verilog)	X	X		O		X	X		O
Génération de Tests	X		X			X			
Génération d’interface JTAG							X		
Informations sur le jeu d’instruction			X	X	X	X	X	X	
Informations sur la structure	X	X			X	X	X	X	
Informations sur la mémoire					O	X	X		

Dans [14], on retrouve l’utilisation de LISA pour la conception de filtres Retinex dédiés à effectuer la correction d’images acquises dans de mauvaises conditions de luminosité. L’avantage de choisir un langage ADL est perçu par l’utilisation d’une architecture non conventionnelle à 7 pipelines offrant un bon compromis entre la complexité et la performance. Au delà du degré de liberté que nous procure ces méthodologies de conception on retrouve généralement un environnement de travail permettant de simuler et de debugger les architectures dédiées crées.

On remarque dans le tableau 1.1 que le langage LISA comparativement aux autres existants nous offre des outils et caractéristiques similaires en plus de permettre de

générer une interface JTAG. Le choix de favoriser cet ADL a été fortement influencé par ces caractéristiques.

1.2. Conception avec Processor Designer et le langage LISA

1.2.1. Description générale de l'environnement de conception

Processor Designer est un outil de conception de processeurs, qui permet aux concepteurs peu expérimentés de développer un processeur de haut niveau et de le générer automatiquement en HDL synthétisable.

On retrouve dans les outils de Processor Designer plusieurs *templates* ou architectures de base permettant de développer plusieurs types de processeurs :

- DSP, qui possède un jeu d'instructions optimisées pour les calculs.
- RISC, qui est un processeur à instructions réduites exécutées en un cycle d'horloge chacune.
- SIMD, qui se caractérise par des instructions longues constituées de plusieurs instructions identiques parallélisées.
- VLIW, qui se caractérise par des instructions longues qui sont des agrégats d'instructions courtes indépendantes.

Ces modèles simples permettent d'assurer la compatibilité avec l'*ISS* (Instruction Set Simulator), les outils du logiciel et la synthèse RTL, éliminant la vérification et l'effort de correction nécessaires pour ces différents niveaux d'abstraction (Fig. 1.1). Processor Designer utilise un langage (version actuelle LISA 2.0) permettant de concevoir un processeur à partir d'un ensemble d'instructions que celui-ci pourra traiter. Parmi les

différents outils on retrouve des outils de développement de logiciel, un générateur de code HDL synthétisable (Verilog/Vhdl), un compilateur, un simulateur et enfin un outil permettant le profilage de code C.

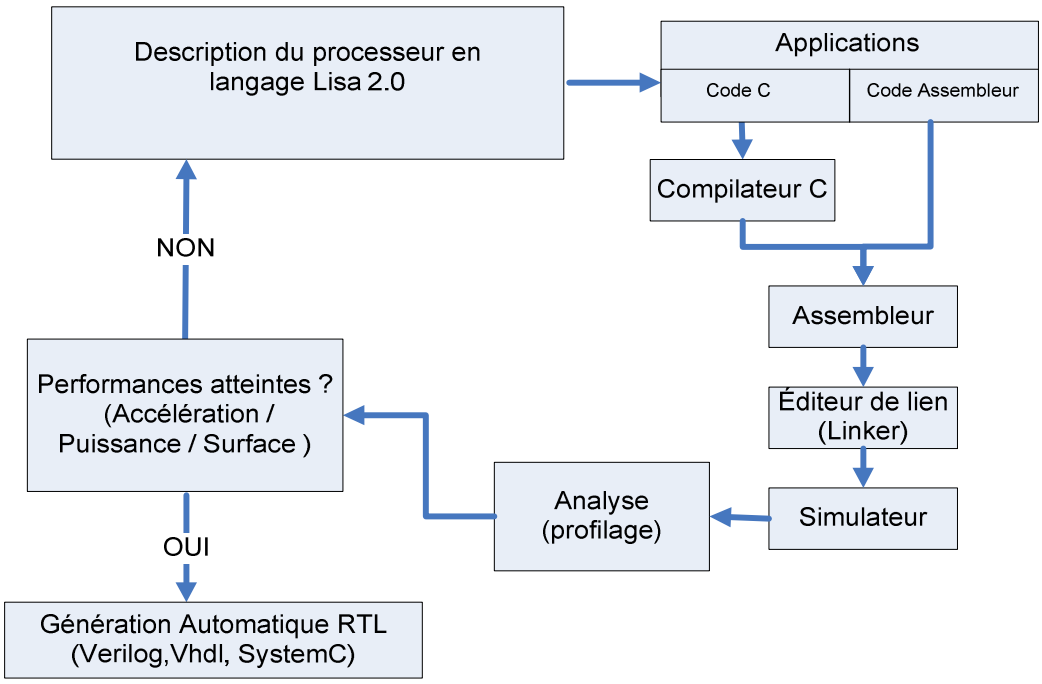


Figure 1.1 : Flot de conception avec Processor Designer

1.2.2. Le langage Lisa 2.0

Le langage Lisa (*Language for Instruction-Set Architectures*) permet une description précise des architectures programmables et de leurs périphériques ou interfaces [16, 17, 18]. La plus récente version disponible, LISA 2.0, se caractérise par la flexibilité de modélisation des architectures, permettant de cibler et de réduire de manière significative les efforts de description. Ce langage soutient un modèle hiérarchique de

description pour une vue claire de l'architecture. Les descriptions d'architectures avec le langage LISA se composent de deux composantes principales :

1. Les ressources, qui modélisent les éléments matériels (bus de données, éléments de mémoire, registres)
2. Les opérations, qui sont les objets de base, qui font abstraction du matériel pour couvrir l'ensemble des instructions, de la synchronisation et du comportement.

Les opérations sont en général organisées comme la structure d'un arbre, ayant pour résultat une suite d'opérations hiérarchique qui décrit l'ensemble des instructions. Elles permettent aux concepteurs d'avoir une vue d'ensemble du comportement structurel du processeur. Pendant la simulation, l'exécution de ces opérations conduit le système dans un nouvel état. Les opérations peuvent être directement traitées ou composées d'une succession d'opérations élémentaires.

On retrouve dans la syntaxe du langage plusieurs sections permettant de définir certaines propriétés ou comportements :

- La section *CODING* est une section qui contient des informations sur le code binaire, d'une instruction (opcode).
- La section *SYNTAX* décrit la syntaxe assembleur, des instructions et de leurs opérandes.
- Les sections *BEHAVIOR* et *EXPRESSION* décrivent des composants du modèle comportemental. Pendant la simulation, l'opération *behavior* est exécutée et elle modifie les valeurs des ressources, ce qui conduit le système dans un nouvel état.

- La section *ACTIVATION* décrit la synchronisation d'autres opérations relativement à l'opération courante.
- La section *DECLARE* contient des déclarations de variables locales et globales.
- La section de *DOCUMENTATION* contient des instructions de documentation

La figure 1.2 présente le squelette d'une architecture en LISA 2.0.

```
RESOURCE
{
 /* Déclaration des registres des processeurs */
 REGISTER uint32 R[0..15];
 /* Déclaration du compteur du programmes */
 PROGRAM_COUNTER uint32 PC;
 /* Déclaration flag */
 REGISTER  bool Z, N, C, V
  /* Déclaration des pipelines du processeur */
 PIPELINE   pipe = { FE ; DC ; EX ; MEM; WB };
}
// définition de l'opération NOP
OPERATION NOP
{
        CODING { 0b0[16] }
        SYNTAX { "NOP" }
        BEHAVIOR
        {
        PC=PC+1;                    // incrémentation du compteur d'instruction
        R[0]=R[10]+ 0x000000ff      // exemple de manipulation des regitres d'un processeur
        printf("Ne fait rien \n");  // Affichage du texte dans la fenêtre de simulation
        }

}
```

Figure 1.2 : Exemple d'opération et de déclaration de ressources en LISA 2.0

La section RESOURCE définit tous les registres de notre architecture. Par analogie avec une description en C, la session RESOURCE définit toutes les variables globales. On y retrouve aussi la définition du pipeline de l'architecture.

Nous présentons dans la section OPERATION de l'exemple de la figure 1.2 un exemple de description d'une instruction. La partie BEHAVIOR qui décrit le comportement de

cette instruction fortement orientée vers le langage de programmation C rend le langage LISA intuitif et facile à utiliser.

1.2.3. Environnement et outils de développement

L'environnement Processor Designer facilite l'interface avec plusieurs outils de développement de processeurs spécialisés. Nous présentons ici la fenêtre principale ainsi que quelques uns de ces outils.

- Fenêtre principale de développement (description textuelle du processeur)

La fenêtre principale de développement est l'interface la plus utilisée dans Processor Designer. Elle permet à l'aide d'une description textuelle, de modifier les instructions, d'attribuer les ressources disponibles, de déterminer le nombre d'étages du processeur, et de manipuler toutes les instructions. Chaque instruction se présente sous forme de plusieurs fichiers d'extension .LISA.

- Instruction Set Designer

La figure 1.3 présente l'outil Instruction Set Designer (ISD) qui permet de manipuler les différentes instructions du processeur. La première colonne de la figure 1.3 nous présente les différentes instructions du processeur.

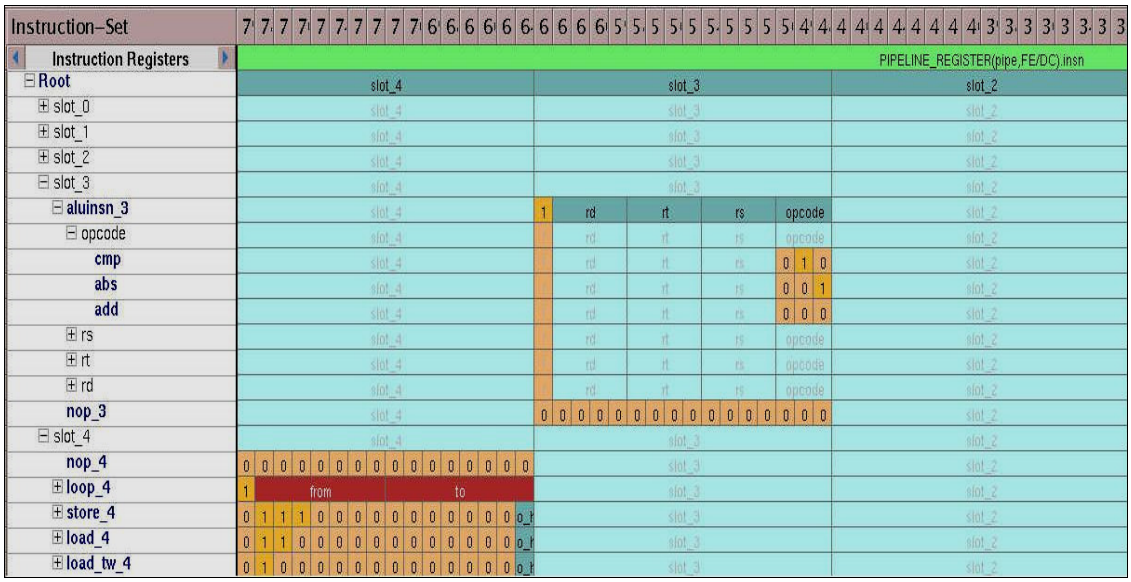


Figure 1.3 : Instruction Set Designer

Les colonnes slot_4 à slot_2 correspondent aux différents registres des pipeline ; respectivement les pipelines FE (Fecth), DC (Decode) et EX (Execute). Si on prend par exemple l'instruction « add », cette dernière utilise un registre de 16 bits situé dans l'étage Decode et ayant trois opérandes (rd, rt et rs) et un opcode (000). Ainsi, cette représentation permet de déterminer le code d'opération ainsi que les opérations effectuées par chacune des instructions. De plus, cette interface donne accès aux registres associés à chaque instruction et permet même de regrouper plusieurs instructions dans un module logique (i.e. ALU). En d'autres termes, l'ISD permet de générer le vocabulaire d'un processeur. Lors des différentes manipulations, l'outil s'occupe de générer le code source LISA correspondant aux changements effectués. Nous travaillons ainsi à un niveau d'abstraction beaucoup plus élevé que le mode texte.

- **Compiler Generator**

Cet outil permet de générer un compilateur spécialisé qui permet d'exploiter notre processeur avec du code C/C++. Dans cette partie, nous retrouvons aussi un grand degré de liberté. Les règles qui permettent de faire un lien entre le code C/C++ et notre architecture peuvent être éditées, modifiées et exportées. Ces règles permettent d'une part de transformer le code C en code assembleur, et d'autre part de générer des fichiers qui seront réutilisés par le ‘‘debugger’’.

Les règles de base disponibles lors de la création d'un projet permettent d'effectuer des opérations telles le branchement, les opérations arithmétiques, les appels de fonction ou encore des opérations booléennes. C'est dans cette partie de l'environnement que nous avons ajouté des règles pour que le compilateur reconnaisse nos instructions spécialisées conçues. Ces règles décrivent la syntaxe des instructions, les ressources utilisées et le temps d'exécution des différentes instructions.

- **Autres outils**

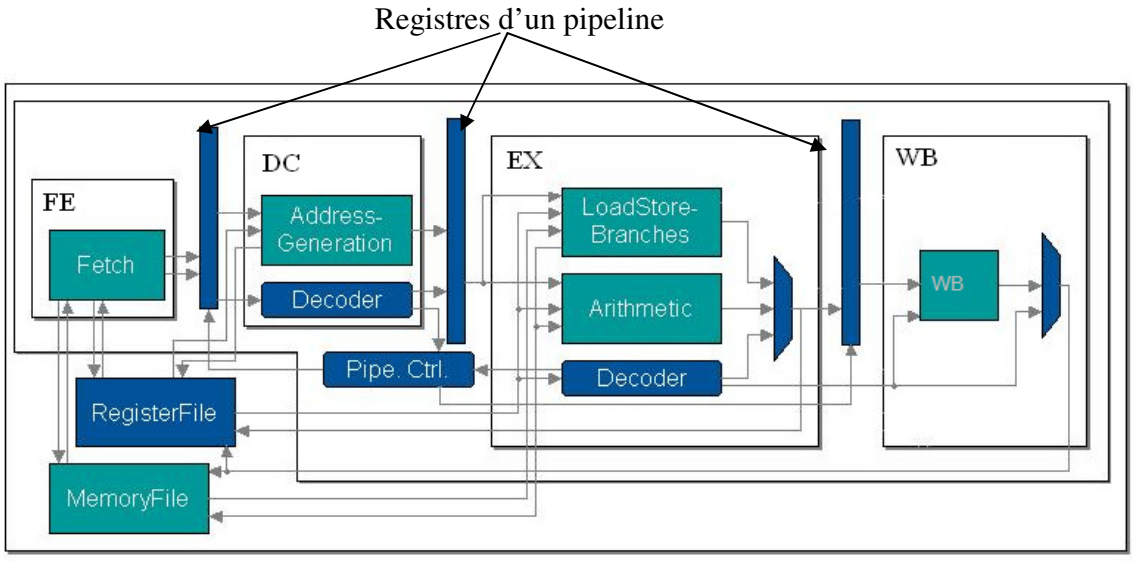
Il existe plusieurs autres outils qui permettent de valider l'architecture conçue. Nous retrouvons un ‘‘debugger’’, un simulateur, un outil de profilage, un outil de désassemblage. On retrouve aussi des outils permettant de visualiser l'attribution des ressources, l'attribution des instructions ou le nombre d'étages présents sur le processeur. Les résultats sont conservés sous forme d'un tableau qui se remplit au fur et à mesure de la simulation.

1.2.4. Architectures de base sous Processor Designer

Lorsqu'on commence un projet sous Processor Designer, nous avons la possibilité de choisir différents types d'architectures pour concevoir notre processeur. Toutes ces architectures sont des processeurs complets avec un jeu d'instruction de base et ayant quatre blocs représentés à la figure 1.4.

A chaque cycle, une nouvelle instruction est importée dans le module FE. Elle est placée à l'entrée du registre du pipeline FE/DC (représenté entre le bloc FD et DC). Si le pipeline, n'est pas bloqué (stalled), l'instruction sera décodée dans DC le cycle suivant. Par la suite le décodeur extrait l'instruction et la décode, il évalue les opérandes et les adresses, s'il y a lieu, et il passe tout vers le module EX à partir du registre DC/EX.

Il active par la suite les modules correspondants à l'exécution de l'instruction dans EX. Il indique aussi, s'il y'a lieu, au module WB, que l'instruction courante nécessite une donnée résultante de l'exécution de l'instruction précédente.



Legende: FE = Fetch ; DC = Decode ; EX = Execute; WB =Write-Back

Figure 1.4 : Pipeline de l'architecture FE

Le module activé dans EX reçoit les données un cycle plus tard, il les exécute et selon l'instruction, les résultats seront sauvegardés directement en mémoire ou passés vers WB par le registre EX/WB. Pour finir, s'il est activé, le module WB sauvegarde les résultats, il permet aussi un transfert direct des résultats vers DC et EX.

1.2.5. Pipeline et le contrôle des "hazard"

Dans l'exécution normale d'un programme par le processeur, chaque instruction passe dans le pipeline de bout en bout, comme le montre la figure 1.5:

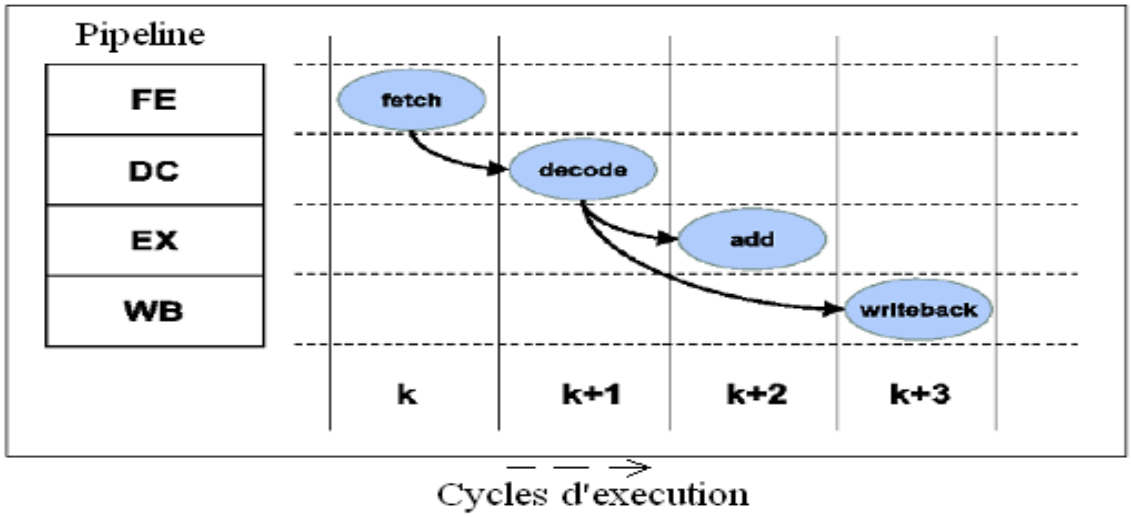


Figure 1.5 : Instructions pipelinées

Pour cela il existe deux fonctionnalités du pipeline qui permettent ce passage :

Execute() : permet l'exécution des opérations activées dans leurs étages correspondants.

Shift() : permet le transfert du contenu d'un registre d'un étage au suivant.

Comme le montre la figure 1.6, si tous les étages du pipeline sont remplis on arrive à chaque cycle à exécuter une instruction différente dans chaque étage.

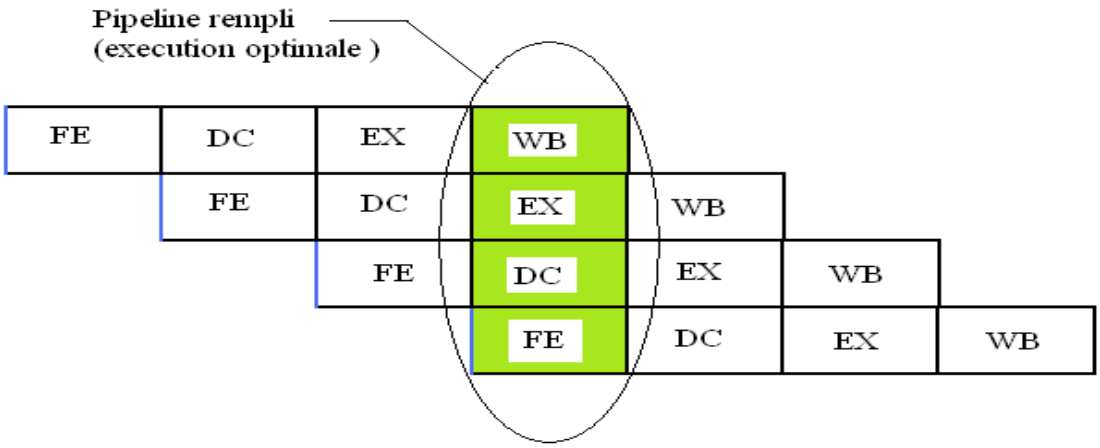


Figure 1.6 : Débit du pipeline

Néanmoins, cette représentation du pipeline ne tient pas compte des dépendances de données entre instructions consécutives. Cela peut introduire des exécutions avec des valeurs résultantes inexactes : on nomme ce phénomène des ‘‘Hazard’’.

Il existe trois classes de hasards qui peuvent se produire durant l’exécution d’un programme donné :

- "Hazard" structurels : dus aux conflits qui se produisent si les ressources matérielles ne supportent pas toutes les combinaisons des exécutions simultanées dans les différents étages du pipeline.
- "Hazard" de données : dus au fait que l’exécution d’une instruction dépend des données résultantes par l’instruction qui la précède.
- "Hazard" de contrôle : dus aux instructions qui changent le compteur de programme comme les branches.

Deux instructions permettent le contrôle des "Hazard" dans le pipeline. L'instruction Stall() qui empêche le décalage des opérations dans l'étage et l'instruction Flush() qui enlève toutes les instructions d'un étage du pipeline.

Examinons l'exemple suivant ou l'instruction JMP fait un branchement vers k1 :

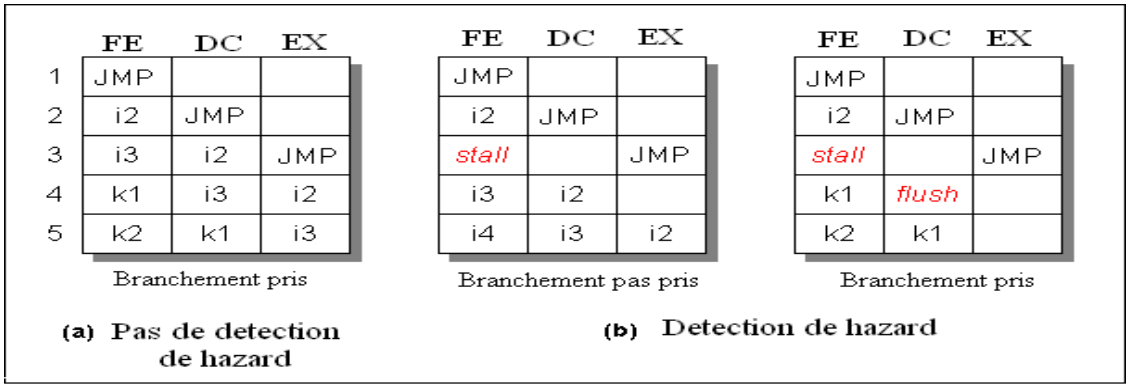


Figure 1.7 : Exemple de détection des "hazard"

On remarque dans la figure 1.7a) que si on n'utilise pas la détection des "Hazard", les deux instructions i2 et i3 passent dans le pipeline et sont tout de même exécutées alors que l'on doit à cette étape évaluer si la fonction de branchement est prise ou pas.

En effet, tels est le cas de 1.7.b qui exécute, lors d'un branchement, l'instruction k1 en ayant précédemment évacué (PIPE.DC.IN.flush()) les instructions i2 et i3. Si le branchement n'est pas pris on perd tout de même un cycle qui permettait à i2 de passer vers l'étage de décodage. Dans tous les cas la détection de hasards nous coûte au moins un cycle de latence.

1.3. Le désentrelacement vidéo

1.3.1. Mise en contexte

L'entrelacement (de l'anglais *Interlaced*) est une technique destinée à réduire le débit nécessaire pour la transmission d'un signal vidéo en ne traitant que la moitié de l'image à chaque fois. Typiquement, une image est décomposée en deux trames consécutives ; une trame paire et une trame impaire. Une trame est une image de demi-résolution verticale qui contient une ligne sur deux de l'image d'origine.

Ce mode d'affichage de la vidéo s'avère bien adapté aux écrans et téléviseurs cathodiques dont le spot va reproduire fidèlement le balayage de chaque trame paire et impaire, la persistance rétinienne faisant le reste. Il s'avère souvent impropre, aux afficheurs numériques (écran VGA, plasmas et LCD, vidéo projecteurs LCD ou DLP…) [19,20].

En effet ces diffuseurs numériques affichent l'image par adressage. Autrement dit, le signal vidéo entrelacé doit être systématiquement converti sous une forme dite "non entrelacée" (on dit aussi "progressif"), puis mis à l'échelle ("scaling") à la résolution native du diffuseur considéré.

Tous les diffuseurs numériques utilisent un signal progressif. Si le signal d'entrée est entrelacé ils intègrent un dispositif de désentrelacement et de mise à l'échelle. L'opération qui consiste à convertir la vidéo entrelacée en vidéo progressive s'appelle le "désentrelacement". Un mauvais désentrelacement entraînera une détérioration de l'image qui se caractérise généralement par des effets de peigne sur les contours des

objets ou encore un effet de traînée sur ces objets en mouvements. Les algorithmes de désentrelacement utilisent des méthodes d'interpolations spatiales et/ou temporelles qui consistent à calculer les lignes manquantes de chaque trame, respectivement, à partir de la trame courante, ou des trames conservées en mémoire. On distingue principalement les algorithmes intra-trame, inter-trames et par compensation de mouvement.

1.3.2. Méthodes de désentrelacement

- **Méthodes intra trame**

Ces méthodes utilisent des techniques d'interpolation spatiales pour calculer les lignes manquantes d'une trame [21, 22]. Elles calculent les pixels manquants de chaque trame à partir de pixels voisins de la trame courante.

Une des méthodes très utilisée dans ce contexte est ELA (Edge-based Line Average). Pour estimer les pixels de la ligne manquante, elle utilise minimalement 3 pixels de la ligne du dessus et 3 pixels de la ligne du dessous [19, 23] :

On détermine tout d'abord selon quelle direction l'interpolation s'effectuera en trouvant le minimum des quantités suivantes :

a = X(k-1,n-1) – X(k+1, n+1) (diagonale de gauche à droite)
b = X(k-1,n) – X(k+1, n) (verticale)
c = X(k-1,n+1) – X(k+1, n-1) (diagonale de droite à gauche)

X représente l'encodage d'une couleur du format RGB ou d'une composantes des triplets Y, Cr, Cb. La figure 1.8 illustre cette méthode.

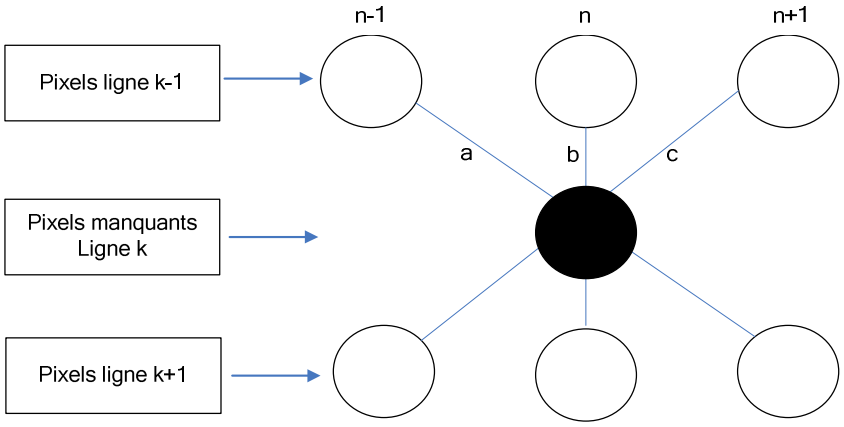


Figure 1.8 : Algorithme ELA « simple »

Une fois le minimum entre a, b et c trouvé, on calcule le pixel manquant en effectuant une moyenne des valeurs des pixels selon la direction d'interpolation déterminée.

On retrouve dans le même ordre d'idée l'utilisation de plusieurs lignes et/ou de plusieurs pixels de la ligne du dessus et du dessous (5, 6, etc...) [24,25]. Dans l'exemple de la figure 1.9 qui suit on utilise cinq pixels de deux lignes entrelacées successives.

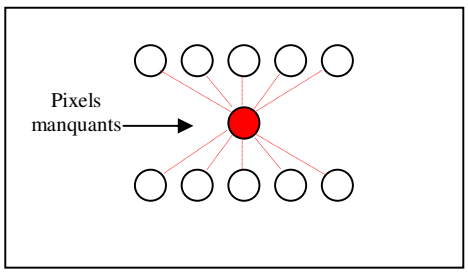


Figure 1.9 : ELA amélioré (cinq pixels)

- Méthodes inter champs

Ces méthodes calculent les pixels manquants de la trame courante à l'aide de trames précédentes ou futures [26]. Il s'agit ici d'interpolations temporelles qui nécessitent de garder en mémoire au moins une trame. On retrouve quelques sous-groupes de ces méthodes à savoir :

- Le doublage de trames (Weave) : on détermine les pixels manquants de la trame courante en utilisant directement ceux de la trame précédente (aucun calcul n'est effectué)
- La moyenne de trames : on détermine les pixels manquants de la trame courante en effectuant la moyenne des pixels des trames précédentes et des trames futures.

La notion de futur est, à toute fin utile, abstraite : on utilise au moins trois trames (mémoires tampons) en effectuant des calculs sur la trame du milieu. Bien entendu une latence est introduite pour produire le résultat final.

Ces méthodes permettent d'obtenir de bons résultats pour des régions statiques de la séquence vidéo. Par contre elles produisent aussi des effets « de peigne » pour des objets en mouvement, détériorant ainsi la qualité de cette vidéo. Dans ce cas de figure, les solutions dite adaptatives permettent d'éviter ce problème.

- Méthodes adaptatives

Ces méthodes combinent les avantages des méthodes intra et inter trames. Elles utilisent des interpolations temporelles dans les zones statiques et des interpolations spatiales dans les zones en mouvement.

La détection de mouvements est ainsi un point crucial de ces différentes méthodes. On retrouve dans la littérature plusieurs algorithmes et méthodes qui se basent sur ce principe [27, 28, 29]. Ces dernières se différencient principalement par les méthodes de détection de mouvements utilisées.

- Méthodes à compensation de mouvements

Les méthodes de compensation de mouvements sont des versions plus complexes et améliorées des méthodes dites adaptatives [21, 30, 31]. Elles effectuent en plus des interpolations temporelles dans des zones statiques. Pour ce faire, elles introduisent le concept d'estimateur de mouvements qui calcule le vecteur de déplacement d'un pixel, d'un bloc de pixels ou encore d'un objet complet par rapport à une ou plusieurs trames (précédentes et/ou futures). On retrouve aussi dans ces méthodes la notion d'appariement de bloc (*block matching*).

1.3.3. Vue globale de l'ensemble des méthodes

Un tableau récapitulatif des différents types de méthodes est représenté au tableau 1.2.

Tableau 1.2 : Méthodes de désentrelacement

Méthode de désentrelacement	Qualité de la méthode	Performance (complexité de calcul)	Mémoire requise
Intra trame	Qualité pauvre	Faible	Minimum de 2 lignes
Inter trames	Qualité pauvre dans les zones en mouvement	Moyenne	Une trame (minimum)

Méthodes adaptive	Bonne qualité (dépend essentiellement de l'algorithme de détection de mouvement utilisé)	Élevée	Une à plusieurs trames
Compensation de mouvement	Très bonne qualité (dépend de l'estimateur de mouvement)	Très élevée	Une à plusieurs trames

1.4. Conclusion

Les méthodologies de conception de processeurs dédiés ont été présentées dans cette partie du mémoire. Cette étape est une entrée en matière qui circonscrit notre sujet de recherche. Après analyse des différents type de processeurs, nous avons opté pour la conception de processeurs dédiés car ils nous offrent une piste de solution favorable pour vaincre les divers défis que soulève le traitement vidéo. Ces défis se résument à des contraintes difficiles à satisfaires en ressources utilisées, vitesse de calcul et bande passante. De ces contraintes est né le désir de concevoir des processeurs dédiés flexibles et performants, nous conduisant à l'utilisation de langages ADL.

Une revue de ces méthodologies de conception basées sur une description complète d'architecture nous a fait opter pour le langage LISA 2.0 qui nous offrait au début de ces travaux les meilleurs niveaux de performance. Par la suite, ce langage a été présenté ainsi que l'environnement de développement Processor Designer de la société Coware. Pour finir, nous avons présenté le mécanisme d'entrelacement en traitement d'image, ainsi que les différentes méthodologies de désentrelacement qui permettent un affichage amélioré des séquences vidéo entrelacées.

CHAPITRE 2. MÉTHODOLOGIES DE CONCEPTION PROPOSÉES POUR L'IMPLEMENTATION D'ALGORITHMES DE TRAITEMENT VIDÉO TEMPS RÉEL

Ce deuxième chapitre présente une vue d'ensemble de notre système implémenté et de l'exploration architecturale s'y rattachant. Le souci principal de la première section est de concevoir un environnement de test complet et modulaire comprenant un diffuseur et un décodeur vidéo, des modules de traitement et un module d'affichage des résultats sur un écran VGA.

Dans la seconde section nous nous concentrons sur deux méthodologies innovatrices pour la conception de processeurs dédiés à partir de langage ADL. La première méthodologie démarre avec un processeur RISC et repose sur l'ajout et le raffinement d'instructions spécialisées. Une motivation principale de cette méthodologie est de conserver une description algorithmique de haut niveau avec le langage C. La deuxième méthodologie reprend les améliorations obtenues à l'aide de la première et intègre la notion de parallélisme des instructions. Le support de départ est une architecture VLIW.

2.1. Vue globale du système

Le système d'acquisition de la séquence vidéo est connecté directement à une planchette de développement XUV2P de Digilent inc. comprenant un FPGA XC2VP30.

Ce système d'acquisition est un décodeur vidéo (VDEC1) de la compagnie *Digilent* [32].

Ce décodeur vidéo possède plusieurs entrées (composite, SVGA) qui lui permettent de recevoir le signal vidéo entrelacé provenant d'un lecteur DVD standard. Nous présentons à la figure 2.1 le schéma bloc du système au complet.

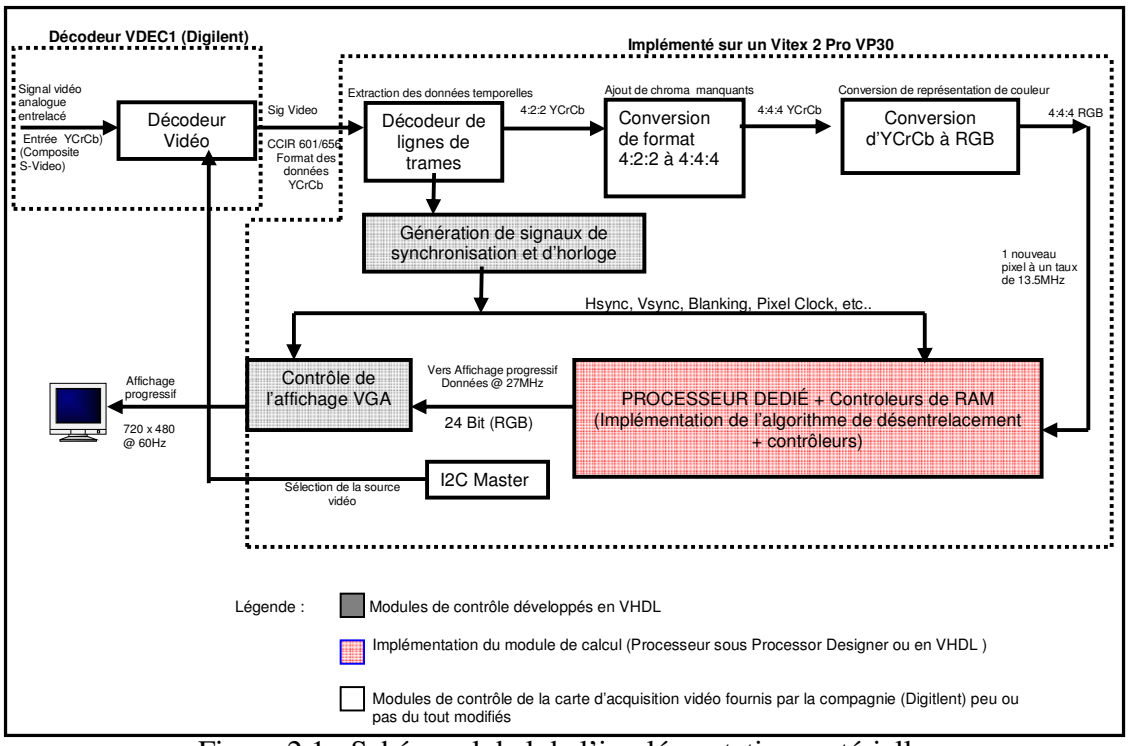


Figure 2.1 : Schéma global de l'implémentation matérielle

Le système se compose des blocs suivants :

- Un Décodeur Vidéo (VDEC1) qui est basé sur trois convertisseurs analogiques-numériques (54 MHz, 10 Bits) ; cette carte détecte automatiquement sur son entrée les signaux vidéo analogiques standard et les transforme sous la forme numérique YCrCb 4:2:2 entrelacée (720x480 dans notre cas). Pour des fins de test, on utilise un lecteur DVD pour alimenter le décodeur qui alimente à son tour

notre système de traitement vidéo. Le signal vidéo généré par le DVD est sous forme entrelacée 4:2:2.

- Un décodeur de lignes de trames qui détermine si nous avons une trame avec des lignes paires ou impaires.
- Un convertisseur de format 4 :2 :2 en 4 :4 :4 . Pour le format 4 :2 :2 nous avons pour 4 pixels successifs 4 Y, 2 Cr et 2 Cb et 4 Y, 4 Cr et 4Cb pour le format 4 :4 :4.
- Un convertisseur de format vidéo d'YCrCb en RGB.
- Un générateur de signaux d'horloge et de synchronisation qui nous fournit les différentes horloges de synchronisation des pixels
- Un module de traitement d'images qui effectue du doublage de ligne
- Et enfin un contrôleur d'affichage qui permet d'afficher le résultat sur un écran VGA dans un format 720x480 progressif (RGB).

Ces différents modules existaient sous forme de fichiers VHDL et Verilog fournis avec le décodeur vidéo. Nous les avons par la suite modifiés comme la figure 2.1 l'indique (les blocs en gris).

Les différentes fréquences que l'on retrouve à savoir 13,5MHz et 27 MHz permettent de respecter la norme NTSC de 60 images par seconde.

Dans cette norme, la séquence vidéo est entrelacée et elle est de résolution de 720x480. La séquence comprend 60 images par seconde pour une résolution réelle de 858x525. En ajoutant les pixels de synchronisation, cela correspond au débit classique de :

$$F \approx 858 \times 525(pixels/image) \times 60(images/\sec) \approx 27MHz$$

Ainsi pour une trame entrelacée, nous avons une fréquence de nouveaux pixels de 13.5MHz par seconde.

2.2. Exploration architecturale

Le but de l'exploration architecturale est d'avoir une réflexion sur la manière d'implémenter notre système. Le décodeur fourni par la compagnie Digilent, effectue le désentrelacement en effectuant du doublage de lignes. Notre système devra quant à lui effectuer le désentrelacement à l'aide d'un algorithme ELA.

En effet, pour concevoir un module de désentrelacement, nous nous sommes concentrés sur une méthode simple de désentrelacement à savoir l'algorithme ELA. Néanmoins, malgré la faible complexité de l'algorithme lui-même, il a fallu déterminer le type d'architecture qui permet d'interfacer le processeur aux systèmes d'acquisition et d'affichage. Chacune des architectures présentées ci-dessous a été étudiée et analysée afin de déterminer les avantages et les inconvénients de chacune. Au total, trois types d'architectures ont été analysés. Chacun d'elle comporte des avantages et des inconvénients. Le but ultime est de trouver le meilleur compromis possible avec les contraintes actuelles qui sont énumérées ci-après :

- Traiter les données en temps réel (Format NTCS : 60 images par seconde)
- Ne pas excéder les ressources du matériel utilisé (FPGA Virtex2Pro Vp30, Fmax 100MHz)
- Développer le système de façon à ce que d'autres algorithmes de mêmes type puissent être facilement implantables (aspect générique du module).

- Flexibilité maximale, permettant de migrer facilement par la suite vers un système de traitement vidéo différent i.e. compensation de mouvement.
- Architecture simple de développement

2.2.1. Architecture centralisée

Cette première architecture de la figure 2.2 comporte un processeur qui effectue tous les traitements nécessaires et une RAM permettant de stocker les données.

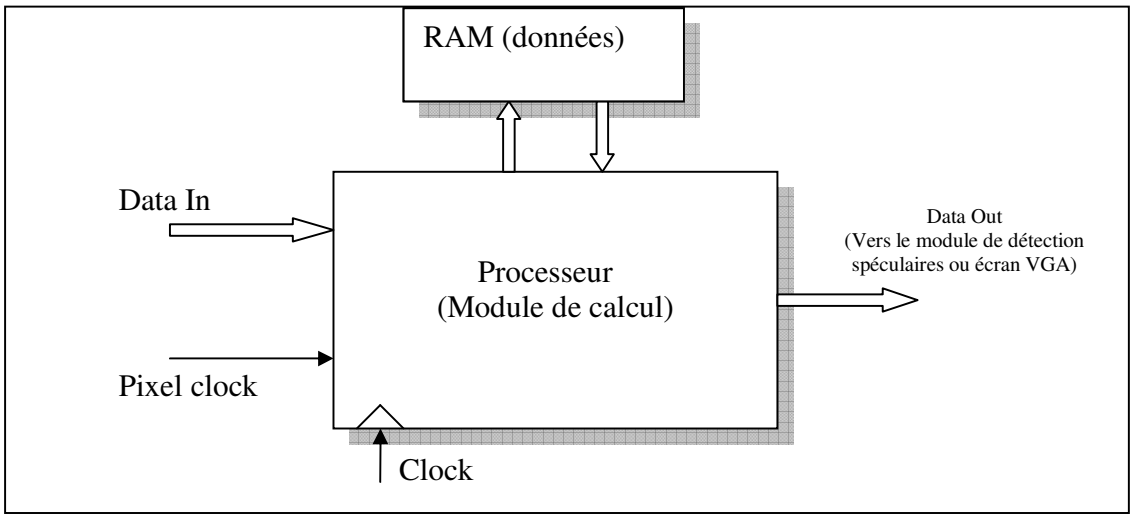


Figure 2.2 : Diagramme de l'architecture centralisée

Cette architecture reçoit directement les données à l'entrée du décodeur vidéo sans aucune manipulation. Le calculateur prend alors ces données et les place dans les éléments de mémoire (RAM). Lorsqu'il y a suffisamment de données en mémoire pour commencer le traitement des données (le calcul des pixels manquants pour chacune des trames), il reprend les données en mémoire et effectue ses calculs. Ensuite, il doit replacer les résultats dans la mémoire. Par contre, durant ces calculs, il doit s'assurer que

les données qui arrivent à l'entrée sont bel et bien récupérées pour ne rien perdre. C'est pour une raison de synchronisation, lors de la récupération des données, que le signal pixel clock (horloge de synchronisation des pixels d'entrée) est à l'entrée du module de contrôle, agissant ainsi comme déclencheur de début de traitement

Le module de calcul doit aussi placer les pixels à la sortie afin de les acheminer vers l'affichage. Un des avantages d'une telle architecture est sa simplicité car elle ne comporte aucun module de synchronisation.

Par contre, cette architecture ne respecterait vraisemblablement pas toutes les contraintes de temps, car il faut impérativement dans le laps de temps déterminé par l'horloge des pixels de 13,5MHz (≈74ns) effectuer tous les traitements nécessaires pour le désentrelacement. Cette situation s'avère très ardue quand on prend en compte le fait que les pixels doivent être affichés à un taux de 27 MHz et que la fréquence maximale du FPGA choisi est de 100 MHz (10ns). Il paraît alors évident que la contrainte de temps ne sera pas respectée : les six accès mémoire pour récupérer les six pixels (3 de la ligne de dessus, 3 de la ligne en dessous), demandent chacun minimalement 2 coups d'horloge. Ce qui requiert en terme de temps 12ns (1ns×6×2=12ns) juste pour accéder aux 6 pixels (sans même prendre en compte le temps de calcul du pixel manquant). Cette solution n'est donc pas envisageable avec un processeur embarqué conventionnel. De plus, développer un seul module responsable de toutes les opérations est très complexe.

2.2.2. Architecture de type « Data Flow »

Dans cette deuxième architecture de la figure 2.3, le processeur permettant d'effectuer le désentrelacement n'effectue que cette tâche : il a en tout temps deux lignes en entrée et deux en sortie. Les contrôleurs d'entrée et de sortie sont à toutes fins utiles des multiplexeurs qui permettent d'acheminer les données au module de calcul et de récupérer les données traitées pour l'affichage.

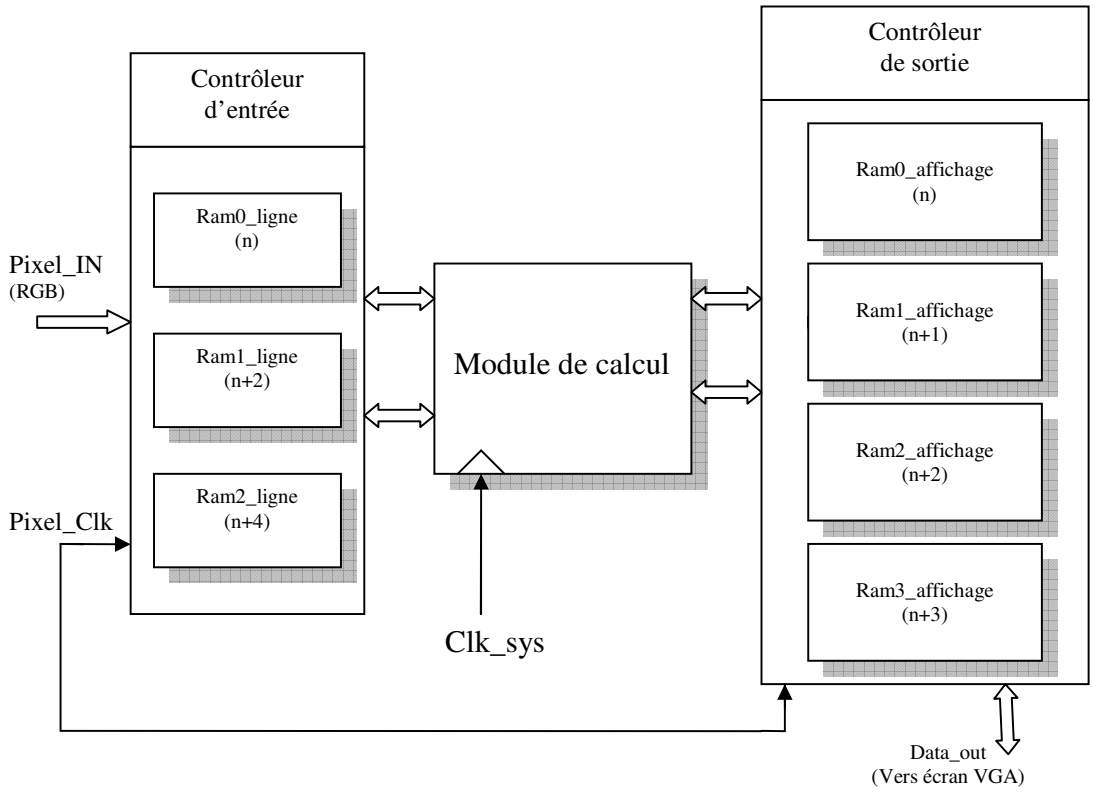


Figure 2.3 : Diagramme de l'architecture de type « Data Flow »

Cette architecture est composée de 3 RAM d'entrée et 4 RAM de sortie. Chacune de ces RAM contient tous les pixels d'une ligne. En entrée, pendant que l'une des trois RAM se

fait remplir par le contrôleur d'entrée, le module de calcul se sert des deux autres, qui sont pleines, pour effectuer ses calculs. Il placera immédiatement dans les RAM de sortie les données récupérées en entrée et celles calculées.

Par exemple, pour une trame impaire, si on est en train de recevoir la 3^{ième} ligne correspondant à la 5ième ligne d'une image complète, le module de calcul accède aux RAM 1 et 2 qui contiennent respectivement les lignes 1 et 3 et il écrit dans les RAM 1 et 2 au niveau de l'affichage.

Tout d'abord, le temps à respecter ici devra être au maximum le temps de remplissage d'une ligne (1/13,5 MHZ× Nombre de pixels d'une ligne). Le fait d'ajouter un ou plusieurs tampons de lignes nous permet de nous donner du temps pour effectuer tous les calculs requis. Par contre, le temps de latence avant d'obtenir la première sortie valide croît avec la profondeur des tampons.

Les avantages d'une telle architecture sont la simplicité et la modularité. En effet, en séparant le traitement en trois blocs, on peut considérer que le problème global est scindé en trois problèmes indépendants : la récupération des données, le traitement et la sortie des données valides. Le temps de conception et de vérification du système est ainsi diminué. De plus, il sera facile de changer le module de calcul pour pouvoir effectuer d'autres algorithmes plus complexes.

Dans ce même ordre d'idées, le concept de tampons d'entrée-sortie pourrait s'appliquer pour des éléments de mémoire qui contiendront une trame complète au lieu d'une ligne. Pendant qu'on remplit une trame complète, le module de calcul traite les données des

deux autres trames, et produit deux trames en sortie. Une limite concrète avec cette architecture découle de la mémoire disponible sur le FPGA.

2.2.3. Architecture modulaire avec arbitre

La dernière architecture de la figure 2.4 est une architecture qui est un compromis entre les deux architectures précédentes. Cependant, elle comporte une RAM qui serait externe au FPGA. Ceci est souvent le cas lorsqu'on veut travailler sur plusieurs trames nécessitant des capacités de mémoire excédant la capacité maximale de mémoire interne du FPGA.

Les contrôleurs d'entrée et de sortie contiendraient une logique plus complexe et seraient considérés comme coprocesseurs. Pour finir, l'allocation de la ressource mémoire serait effectuée par un arbitre.

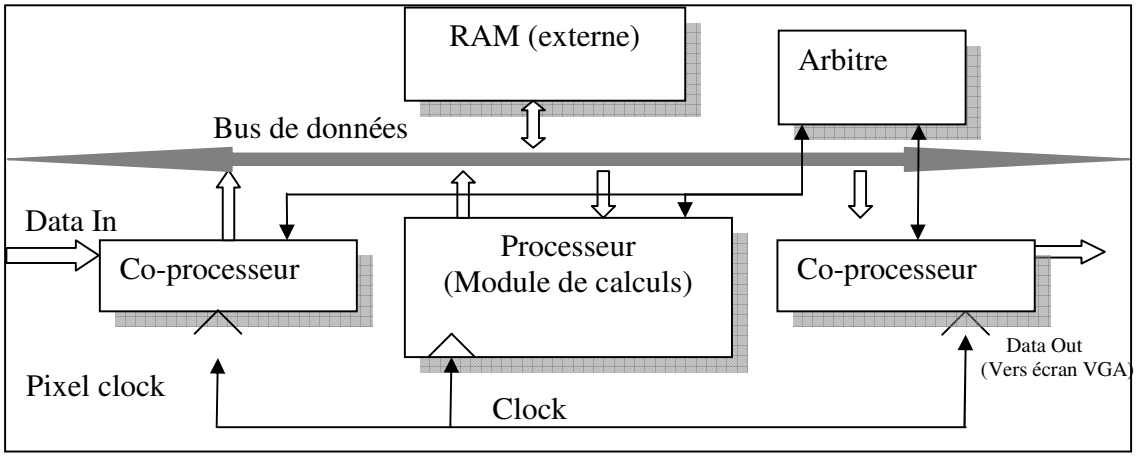


Figure 2.4 : Diagramme de l'architecture de type modulaire avec arbitre

L'acquisition et l'affichage des données seraient effectuées par des coprocesseurs. Les données seraient reçues par le premier coprocesseur et seraient placées en mémoire par

celui-ci. Il est certain qu'à ce niveau, des « Fifo » seraient nécessaires afin de s'assurer que le coprocesseur puisse prendre toutes les données avant de les écrire en mémoire. Le module central de calcul implémenterait comme dans les autres architectures l'algorithme de désentrelacement, en récupérant les données dans la mémoire, et en écrivant les valeurs calculées en mémoire. Finalement, le coprocesseur de sortie demanderait l'accès à la mémoire par le biais de l'arbitre et enverrait les pixels à la sortie du module de désentrelacement. Cette architecture ne devrait pas demander plus de mémoire que nécessaire si la gestion est bien faite. Par contre, le contrôle ainsi que le protocole de communication peuvent être très rapidement des problèmes épineux pour respecter les contraintes de temps.

Néanmoins, une fois le contrôle maîtrisé, ce système peut devenir très performant et flexible pour plusieurs algorithmes de désentrelacement. Selon la taille du bus et par un mode d'adressage judicieux, on pourrait accéder à un bloc de pixels en mémoire.

2.2.4. Choix final de l'architecture

Le choix d'une architecture idéale est complexe. Chacune d'elles comporte des avantages et des inconvénients. Nous avons arrêté notre choix sur l'architecture de type « Data Flow », car à notre avis cette solution offre une complexité moindre. La dernière solution est une solution conventionnelle, mais elle nécessite des modules de gestion de RAM externes ainsi qu'une interface avec le processeur.

2.3. Proposition de méthodologie de conception de processeurs dédiés basés sur du ADL

2.3.1. Conception de processeurs basés sur un processeur RISC

2.3.1.1. En restant à haut niveau (algorithme décrit en C)

La première étape de la méthodologie proposée est d'utiliser un processeur avec un jeu d'instructions élémentaires, décrit avec un ADL dans un environnement comme Processeur Designer. Ce processeur peut être choisi parmi plusieurs architectures telles que RISC, SIMD, VLIW ou DSP. Ces modèles disposent d'une série d'instructions semblable à un processeur d'usage général. Dans notre cas, pour sa faible complexité, nous avons choisi comme point de départ un processeur RISC avec un pipeline de quatre étapes, à savoir FD (fetch), DC (décoder), EX (exécuter) et WB (write back), comme le montre la figure 2.5.

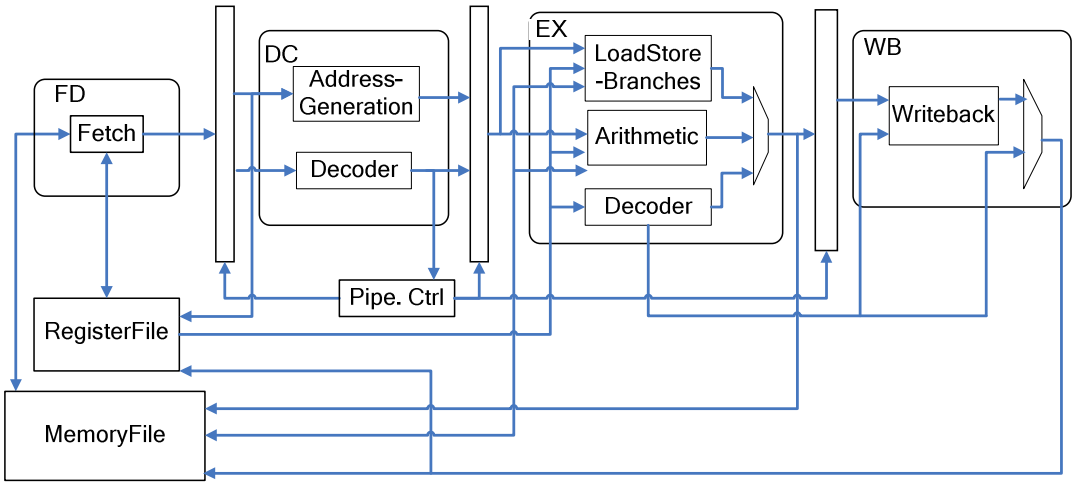


Figure 2.5 : Pipeline du système

La méthode de conception adoptée prend, en entrée, un algorithme codé en C et une architecture de base décrite dans un ADL. Le processus itératif est montré à la figure 2.6. La boucle principale doit être exécutée à plusieurs reprises. Cette boucle correspond grosso modo à la conception traditionnelle de traitement de la configuration des flux, mais cette méthode systématise les différentes mesures qui peuvent être prises. En général, les actions concernent le paramétrage et l'extension du processeur de base. Une cinquième étape vient supprimer les instructions inutilisées.

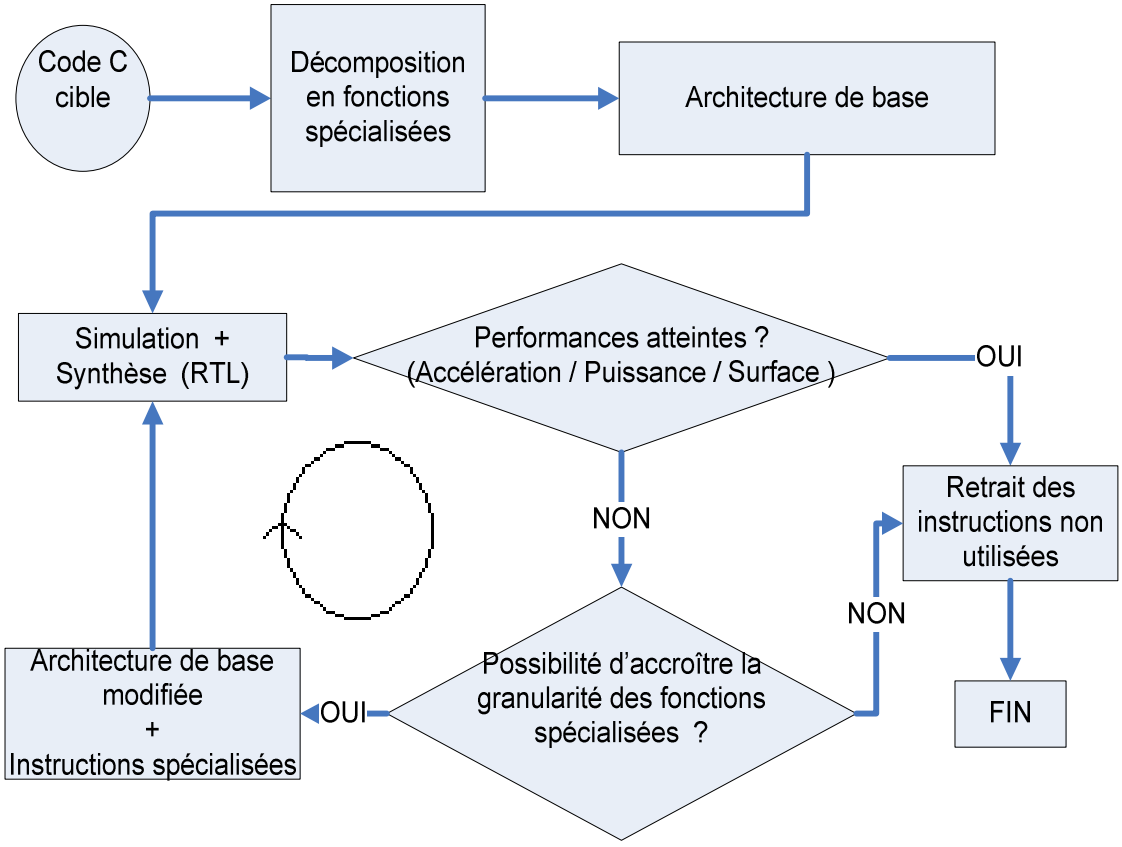


Figure 2.6 : Méthodologie de conception basée sur un processeur RISC

La méthodologie adoptée se décompose de la façon suivante :

A) Décomposition du code C en fonctions spécialisés

Cette première étape nous conduit à regrouper à haut niveau le code source en petites fonctions dans le but de pouvoir par la suite développer du matériel dédié pour les parties de fonctionnalité où c'est justifié. Le matériel dédié est défini dans ce cadre par des instructions spécialisées décrites en langage LISA permettant ainsi d'accélérer le calcul.

Ces fonctions se caractérisent essentiellement par leurs rémanences dans le code en C. Il est à noter qu'à ce niveau, le travail préparatoire de haut niveau, en créant des fonctions, augmente le nombre de cycles d'exécution. En effet lorsque l'on rajoute des fonctions au niveau du code C, des cycles supplémentaires proviennent de l'appel de ces différentes fonctions. Néanmoins la création de ces petites fonctions permet de mettre en évidence l'effort à fournir pour parvenir à nos objectifs.

B) Mise à l'échelle du processeur

Cette étape correspond à une modification de la largeur du bus d'adresse et/ou de données, mais aussi de la largeur des éléments du processeur. Dépendamment des types de traitements, il se pourrait que le processeur de base choisi ne soit pas approprié aux données. Par exemple on pourrait imaginer un processeur devant traiter des données de 8 bits pour un ensemble de valeur contenues sur une plage d'adresse de 10 bits. La modification nécessaire dans ce cas est de changer le bus de données et d'adresses pour respectivement 8 et 10 bits. Toujours dans cet exemple, les différentes fonctions de base

devront être mises aussi à l'échelle (i.e. additionneur 8 bits au lieu de 32). Cette opération n'aurait pas été possible avec d'autres types de méthodologies de conception qui imposent une largeur fixe du bus d'adresses et de données.

C) Augmentation croissante de la granularité des fonctions spécialisées

Il est pertinent d'augmenter la granularité de nos fonctions spécialisées, c'est-à-dire de prendre des blocs plus gros du code source en C et d'instancier des fonctions spécialisées de facto de plus en plus complexes. Cette action peut augmenter le chemin critique du processeur. Une façon de remédier à ce problème est d'ajouter une autre étape de pipeline ou d'augmenter le nombre de cycles de ces instructions spécialisées. Il est crucial, à ce stade, de préserver la fréquence de fonctionnement du processeur.

D) Insertion de registres spécialisés

A ce niveau de notre exploration, si les résultats ne sont toujours pas satisfaisants, nous proposons d'introduire des registres spécialisés qui permettent de minimiser l'utilisation de paramètres dans les appels de fonctions mais aussi de palier à l'inefficacité des registres d'usage général.

En effet lorsqu'on utilise les registres d'usage général (32 au total) et qu'on effectue une opération, le résultat est sauvegardé dans la mémoire des données et si l'on désire par la suite réutiliser ce résultat, il faudra lire les données précédemment écrites. Par contre, en utilisant des registres spécialisés, ces accès à la mémoire de données externes n'ont pas lieu et nous permettent ainsi de sauver des instructions.

E) Retrait d'instructions inutilisées

Pour tirer tous les avantages de cette méthodologie basée sur le langage LISA il est essentiel d'enlever, lorsque nous arrivons au terme de notre exploration architecturale, toutes les instructions inutilisées, autant celle rajoutées que celles du processeur de base. Cette dernière étape nous permet de diminuer la surface occupée par notre processeur, mais aussi de diminuer le chemin critique de celui-ci (décodage d'instructions).

2.3.1.2. Retrait du compilateur (algorithme en assembleur)

Arrivés au terme des étapes de la méthode adoptée, on s'aperçoit qu'en restant à haut niveau, c'est-à-dire en décrivant notre algorithme en C, nous avons moins de contrôle sur la performance du processeur. Prenons l'exemple de l'appel d'une fonction spécialisée effectuant la moyenne entre deux entiers et retournant le résultat.

Soit l'appel suivant : ligne_r_ela[0] = moyenne2(ligne_dessus[0],ligne_dessous[0]);

Le code assembleur généré à partir de cette ligne est :

```
ori r1,r0,ligne_dessus:0x888c
ori r2,r0,ligne_dessous:0x801c
lbu r1,r1,0x0000
lbu r2,r2,0x0000
nop
moyenne2(r1,r1,r2)
sw r1,r0,0x8008
```

Les deux premières lignes sont inutiles si l'on écrit en assembleur directement car nous savons exactement à quelle adresse nous voulons lire nos données.

Nous aurons alors :

```
lui r1,0x888c
lui r2,0x801c
nop
moyenne2 (r1, r1, r2)
sw r1,r0,0x8008
```

La méthodologie de conception basée sur le processeur RISC présentée précédemment est toujours valable. A la fin de celle-ci, on peut tout simplement migrer le code C vers un code assembleur.

2.3.2. Méthodologie de conception basée sur un processeur VLIW

Lorsqu'on veut effectuer une accélération matérielle d'un algorithme exécuté par un processeur, nous sommes confrontés à deux dimensions du problème : la dimension matérielle et la dimension logicielle. La dimension matérielle peut se subdiviser en deux grands blocs : la partie traitements et la partie communications avec le monde extérieur (load/store) ou encore flot et/ou transferts de données. La dimension logicielle représente le langage utilisé pour décrire le traitement et la description elle-même (l'algorithme).

Dans un monde ASIP où coexistent ces deux dimensions, qui nous offrent un compromis entre la performance et la flexibilité, pour obtenir un maximum de performance, il faut effectuer des efforts d'optimisation dans ces deux dimensions. Dans la première méthodologie de la figure 2.6, des efforts d'optimisation ont été effectués dans ces deux dimensions : d'une part, en développant et en intégrant des instructions et des registres spécialisés et d'autre part, en modifiant le code source C pour utiliser ces instructions spécialisées.

Néanmoins, on atteint après plusieurs itérations dans la boucle de la figure 2.7 des plafonds d'accélération. Dans la partie matérielle les efforts investis se concentraient sur les traitements (développement d'instructions spécialisées) sans pour autant toucher à la partie communication (load / strore), devenue par conséquent le goulot d'étranglement.

De plus, dans l'application de la méthodologie, le processeur de base choisi était un processeur 32 bits RISC nous limitant à des transferts de données de 32 bits. Dans la partie logicielle, le fait de rester à haut niveau (description en C) nous laisse à la merci du compilateur limitant ainsi nos performances.

Avec cette seconde méthode, nous proposons de nous attaquer à ces deux dimensions pour accroître nos performances de calculs. Ces deux méthodes peuvent alors être perçues respectivement comme une première itération à haut niveau (raffinement des instructions spécialisées) et une deuxième bas niveau (amélioration du chemin de données et retrait du compilateur). Cette méthode se compose de cinq étapes, montrées à la figure 2.7.

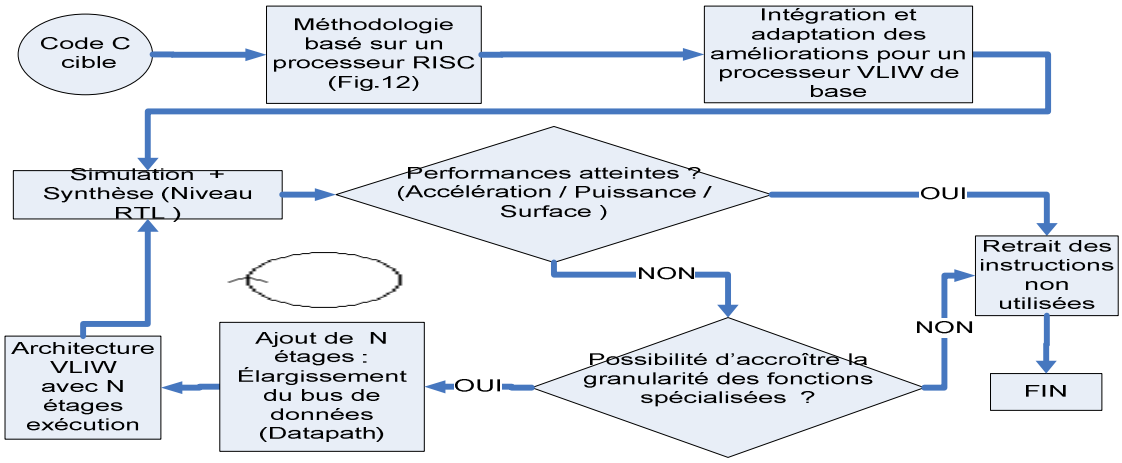


Figure 2.7 : Méthodologie de conception basée sur un processeur VLIW

A) Méthode basée sur un processeur RISC

Cette première étape cruciale peut être perçue comme une étape préparatoire où l'on retrouve le profilage du code C ainsi que la création d'instructions et registres spécialisés. Le choix d'un processeur RISC comme point de départ de la précédente méthode était principalement motivé par sa faible complexité. Arrivés au bout des modifications possibles avantageuses au niveau d'ajouts d'instructions spécialisées, nous avons déduit que pour obtenir de meilleures performances il fallait élargir le bus de données. Augmenter le bus de données en utilisant un processeur VLIW au lieu du RISC nous permet de créer des méga-instructions (ou instructions à plusieurs mots) qui reprennent plusieurs instructions du RISC. D'un point de vue matériel, cela se traduit par plusieurs unités de traitements en parallèle pour l'architecture VLIW comparativement à une seule unité de traitement pour une architecture RISC.

B) Adaptation et intégration des modifications effectuées pour le RISC

Dans cette partie nous réutilisons le travail d'amélioration du processeur préalablement effectué avec le RISC. Nous ajoutons ainsi les instructions et registres spécialisés.

A ce niveau nous pourrons constater si l'ajout des modifications faites pour le RISC améliore les performances. Dans le meilleur des cas, la migration de RISC à VLIW permettra un gain d'accélération d'un facteur de N, N étant le rapport entre le chemin de données du VLIW et celui du RISC. Dans le pire des cas, nous aurons des performances d'exécution égales avec un ‘‘Overhead’’. Tel sera le cas si toutes les instructions du code C ont des dépendances de données.

C) Changement du modèle de programmation

Le changement du modèle de programmation vient répondre à l'autre limite d'accélération : l'inefficacité du code C. En effet, travailler à haut niveau nous limite pour ce qui est des transferts de données internes et externes.

Lors de l'appel d'une fonction ou de la sauvegarde de résultats intermédiaires de calculs, nous avons plusieurs appels en mémoire inutiles. C'est le cas où un résultat est sauvegardé en mémoire et par la suite réutilisé : il en découle une opération *store* suivie d'une opération *load* de la mémoire de données.

Le passage du C en assembleur nous permet de manipuler les registres généraux ainsi que les appels de fonction de manière plus efficace.

D) Vers une architecture hybride SIMD/VLIW

L'élargissement du bus de données conjugué à l'utilisation d'une architecture VLIW permet d'exécuter plusieurs instructions différentes dans le même cycle d'horloge et d'effectuer dans une même instruction plusieurs mêmes traitements en parallèle.

Ce cas correspond à une architecture SIMD (single instruction multiple data) qui peut être perçus comme un cas particulier d'une architecture VLIW.

La figure 2.8 nous présente les implications du passage d'une architecture RISC vers SIMD/VLIW :

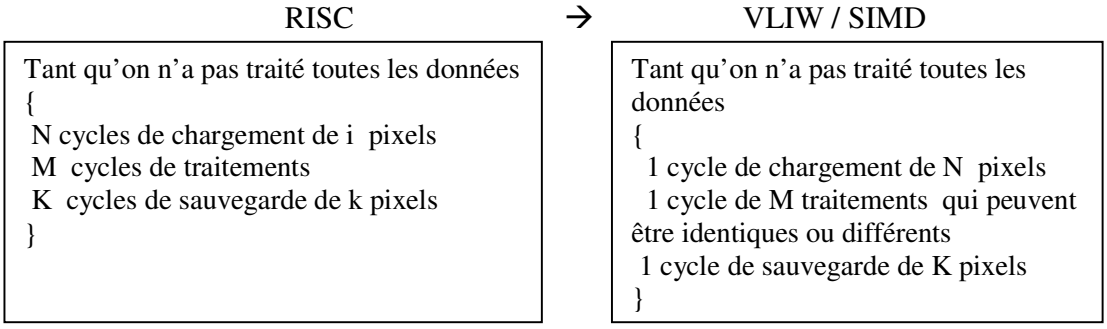


Figure 2.8 : Passage RISC vers VLIW/SIMD

De ces modifications peut résulter une certaine caractérisation du type d'algorithmes dont le temps d'exécution pourrait être amélioré lors du passage de RISC à VLIW. Il faudrait ainsi que les donnés utilisées dans le traitement courant ne soient pas dépendantes des résultats précédents. En d'autres termes, il ne doit pas y avoir de dépendances de données d'un traitement à l'autre afin de pouvoir effectuer plusieurs traitements consécutifs en parallèle au cours du même cycle d'horloge.

E) Nettoyage du processeur

Pour tirer tous les avantages de notre méthodologie il est essentiel d'enlever, lorsque nous arrivons au terme de notre exploration architecturale, toutes les instructions inutilisées. Il faut ainsi supprimer les instructions de base et celles rajoutées au cours du processus de raffinement, mais non inutilisées par la dernière version du processeur. Cette dernière étape nous permet de diminuer la surface occupée par notre processeur mais aussi de diminuer le chemin critique de celui-ci (décodage d'instructions).

2.4. Conclusion

Dans ce chapitre nous avons présenté une vue globale de notre système. Nous sommes partis d'une plateforme de traitement vidéo existante pour nous permettre de développer nos processeurs dédiés dans un environnement de test réel.

Au départ, cette plateforme permettait d'acquérir une séquence vidéo entrelacée d'un lecteur dvd et d'afficher la vidéo désentrelacée résultante sur un écran VGA. La méthode de désentrelacement qui était implémentée était le doublage de ligne qui offre une qualité d'affichage médiocre.

Le but de la partie exploration aura été de trouver une solution modulaire qui nous permette d'implémenter nos différents processeurs développés avec du ADL. Nous avons par la suite exploré deux méthodologies de conception de processeur dédiés à partir de LISA 2.0.

L'algorithme cible auquel peut s'appliquer nos deux méthodes n'a pas été circonscrit. Néanmoins, on peut affirmer que plus le traitement est complexe plus le potentiel d'accélération est élevé.

En effet, dans ces deux méthodes, il s'agit de transposer la complexité logicielle et complexité matérielle par des ajouts d'instructions spécialisées et des modifications architecturales. Plus tard, dans le chapitre 4, nous appliquerons nos méthodes à un algorithme de désentrelacement, l'algorithme ELA.

CHAPITRE 3. IMPLANTION DU SYSTÈME

Dans ce chapitre, nous présentons les résultats de synthèse des différentes parties de notre système. Nous décrivons en détail l'implémentation du système global décrit en VHDL. Cette implémentation matérielle permet de tester le système au complet (Banc d'essai ou ''Test bench'').

3.1. Architecture globale du système

Dans un premier temps, nous avons implémenté notre système à partir d'une description VHDL. Cette approche nous a permis d'obtenir un environnement de test réaliste. Cette partie détaille donc l'implémentation du système de la figure 3.1 que conçues en VHDL.

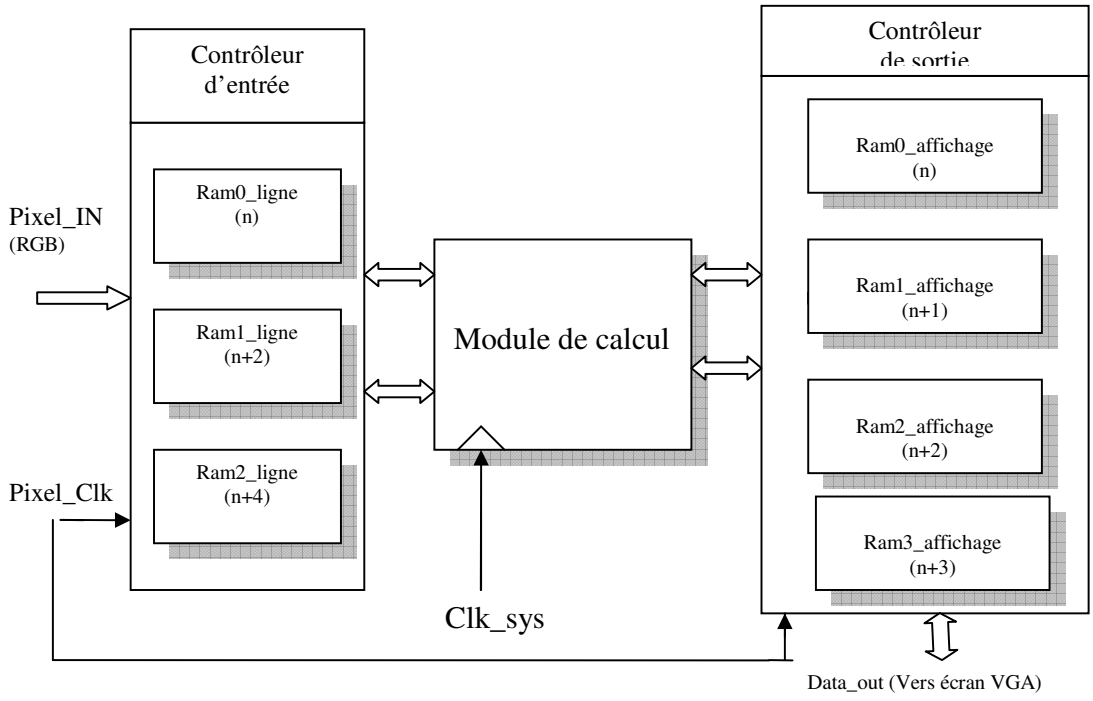


Figure 3.1 : Schéma de l'architecture choisie

Cette architecture, conçue pour implémenter l'algorithme ELA peut être réutilisée pour traiter d'autres types d'algorithmes en modifiant le bloc « module de calcul ». Nous proposons une architecture qui permet à tout moment au module de calcul d'accéder à deux éléments de mémoire en entrée et en sortie. Ci-dessous nous détaillons en détail les signaux importants de notre architecture:

- **« Clk_sys »** est l'horloge du système implémenté, typiquement nous avons une fréquence de 100 MHz disponible sur notre plaquette de développement (cf. figure 3.1).
- **« Pixel_Clk »** est une horloge de synchronisation qui arrive avec la séquence vidéo. Elle est envoyée vers les deux contrôleurs et sa valeur est de 27 MHz. Cette valeur est fournie par le décodeur vidéo de notre système.
- **$Pixel_{IN}$**, Correspond à la valeur du pixel d'entrée. Pour le signal vidéo 4:2:2 en format NTSC, entrelacé, nous avons une nouvelle valeur de la composante (Y ou Cr, ou Cb) à une fréquence de 27Hz. Après le passage du format 4:2:2 à 4:4:4 on obtient un nouveau format de représentation de couleurs (YCrCb =24 bits) à 13,5MHz. Ce vecteur est transformé par la suite en format RGB et acheminé au contrôleur d'entrée.
- **Contrôleur d'entrée :** comprend un contrôleur et trois « RAM ». Chaque RAM est remplie avec une seule ligne de l'image en cours. Pour cela le contrôleur détecte qu'une nouvelle donnée est disponible et remplit une des trois RAM de façon cyclique. En plus, il doit indiquer au module de calcul lesquelles des deux autres sont disponibles pour pouvoir commencer son traitement.

- **Data_out :** correspond au signal de sortie « désentrelacé », 8 bits par couleur pour chaque pixel à 27 MHz (On effectue un traitement identique pour chaque couleur)
- **Le module de calcul :** C'est le module le plus important, il doit exécuter toutes les opérations de calcul et d'accès en respectant les contraintes de temps réel. Son rôle consiste à lire les deux RAM en entrée (1 et 3 par exemple), puis à calculer la ligne manquante (2) pour finalement enregistrer les résultats dans les deux FIFO de sortie (1 et 2 dans ce cas). Ce cas correspond à l'étape 3 du tableau 3.1.
- **Contrôleur sortie :** En même temps que le module de calcul remplit deux des quatre RAM, ce contrôleur achemine les données contenues dans les deux autres RAM de sortie vers l'affichage VGA en sortie (data_out).

3.2. Contrôleurs d'entrée/sortie (Gestion de RAM externes)

- Le contrôleur d'entrée :

Il permet d'écrire les pixels entrants dans une des trois RAM d'entrée et d'acheminer les données contenues dans les deux autres RAM au module de calcul. De façon concrète, en VHDL, nous avons décrit ce contrôleur comme une machine à états possédant 3 états (annexe 1):

- Etat 0 : Les pixels reçus sont écrit dans la RAM0 (d'entrée) et les signaux de la RAM1 et de la RAM2 sont contrôlés par le module de calcul.

- État 1 : Les pixels reçus sont écrit dans la RAM1 (d'entrée) et les signaux de la RAM0 et de la RAM2 sont contrôlés par le module de calcul.
- État 2 : Les pixels reçus sont écrit dans la RAM2 (d'entrée) et les signaux de la RAM0 et de la RAM1 sont contrôlés par le module de calcul.

• Le contrôleur de sortie

Dans le même ordre d'idées que le traitement du contrôleur d'entrée, le contrôleur de sortie permet d'une part au module de calcul de contrôler les entrées/sorties de deux des quatre RAM de sorties et d'autre part d'effectuer l'affichage des deux autres RAM.

Nous avons aussi décrit notre contrôleur comme une machine à deux états.

- État 0 : La RAM2 et la RAM3 sont lues pour effectuer l'affichage. Les signaux de la RAM0 et de la RAM1 sont contrôlés par le module de calcul.
- État 1: La RAM0 et la RAM1 sont lues pour effectuer l'affichage. Les signaux de la RAM2 et de la RAM3 sont contrôlés par le module de calcul.

Il est pertinent de noter que l'assignation de la sortie s'effectue à l'aide de deux autres processus : le premier processus permet de déterminer la fin de l'affichage d'une ligne, le deuxième d'assigner la sortie (rgb_out).

En résumé, lorsqu'on reçoit une nouvelle ligne de la trame courante :

• On écrit cette nouvelle ligne dans une des RAM d'entrée (à 13,5 MHz).

• On calcule une ligne manquante en utilisant les deux autres RAM d'entrée précédemment remplies et on écrit dans deux RAM de sortie la ligne calculée et la ligne de dessus (à 100MHz).

- On sort les valeurs traitées (rgb_out), en lisant les 2 autres RAM de sortie contenant une ligne calculée et la ligne de dessus correspondant. Ces RAM étant remplies à l'étape précédente (affichage à 27MHz).

Si on prend en exemple le traitement d'une trame impaire entrelacée, nous aurions à chaque réception d'une nouvelle ligne la configuration présentée au tableau 3.1.

Tableau 3.1 : Processus d'assignation des RAM dans le cas d'une lecture de trame impaire

	#RAM	**Étape 1**	**Étape 2**	**Étape 3**	**Étape 4**	**Étape 5**
RAM d'entrée (contrôleur d'entrée)	**RAM0_ligne**	**Ligne 1 (E_N)**	**Ligne 1 (L_C2)**	**Ligne 1 (L_C1)**	**Ligne 7 (E_N)**	**Ligne 7 (L_C2)**
	RAM1_ligne	**X (L_C1)**	**Ligne 3 (E_N)**	**Ligne 3 (L_C2)**	**Ligne 3 (L_C1)**	**Ligne 9 (E_N)**
	RAM2_ligne	**X (L_C2)**	**X (L_C1)**	**Ligne 5 (E_N)**	**Ligne 5 (L_C2)**	**Ligne 5 (L_C1)**
RAM de sortie (contrôleur d'entrée)	**RAM0_affichage**	**X (E_1)**	**X (A1)**	**Ligne 1 (E_1)**	**Ligne 1 (A1)**	**Ligne 5 (E_1)**
	RAM1_affichage	**X (E_2)**	**X (A2)**	**Ligne 2 (E_2)**	**Ligne 2 (A2)**	**Ligne 6 (E_2)**
	RAM2_affichage	**X (A 1)**	**X (E_1)**	**X (A1)**	**Ligne 3 (E_1)**	**Ligne 3 (A1)**
	RAM3_affichage	**X (A2)**	**X (E_2)**	**X (A2)**	**Ligne 4 (E_2)**	**Ligne 4 (A2)**

Légende :
E_N=écriture d'une nouvelle ligne dans une des RAM d'entrée

L_C1=lecture de la ligne de dessus dans une des RAM d'entrée,

L_C2=lecture de la ligne d'en dessous dans une des RAM d'entrée

E_1 = écriture de la ligne de dessus dans une des RAM de sortie

E_2 = écriture de la ligne calculée dans une des RAM de sortie

A1 = Le contrôleur de sortie achemine les données lues vers la sortie

A2 = Le contrôleur de sortie achemine les données calculées ou lues vers la sortie.

On remarque ici que notre système sort la première ligne valide à l'étape 4, on a ainsi un temps de latence correspond au remplissage de 3 lignes.

Le flot de données est tel qu'à chaque fois qu'on écrit une ligne à une fréquence de 13,5 MHz, on en affiche deux à 27 MHz.

Le taux de production de trames désentrelacées est alors égal au taux de lecture de trames entrelacées : on produit deux fois plus d'information à une fréquence double (comparativement à la lecture). On a ainsi un taux de trame constant et qui respecte les contraintes de temps réel.

3.3. Module de traitement développé en matériel (VHDL)

Le module de calcul, permettant d'effectuer le désentrelacement a été décrit par une machine à état en VHDL. Au début du traitement, on lit trois pixels de la ligne de dessus et trois pixels de la ligne en dessous. On effectue le calcul de la direction d'interpolation puis on calcule le pixel manquant selon cette direction. Par la suite, jusqu'à la fin de la ligne on effectue une lecture de 2 pixels à chaque fois en réutilisant les valeurs déjà lues, on a un effet de fenêtre coulissante illustrée à la figure 3.2:

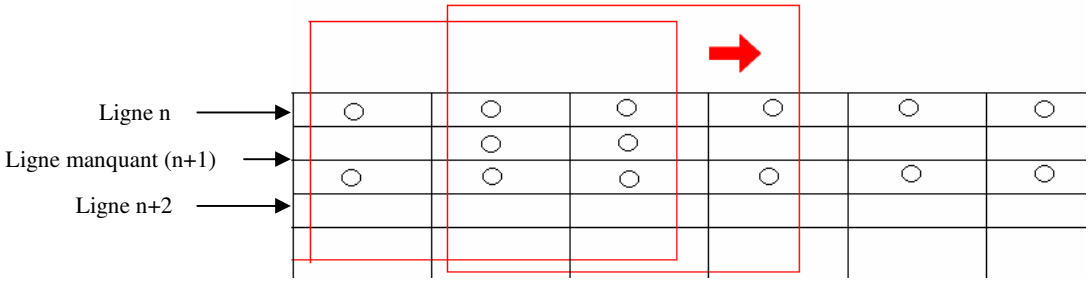


Figure 3.2 : Processus de parcours des RAM d'entrée

Ce processus est représenté ci-dessous sous forme d'étapes successives à la figure 3.3.

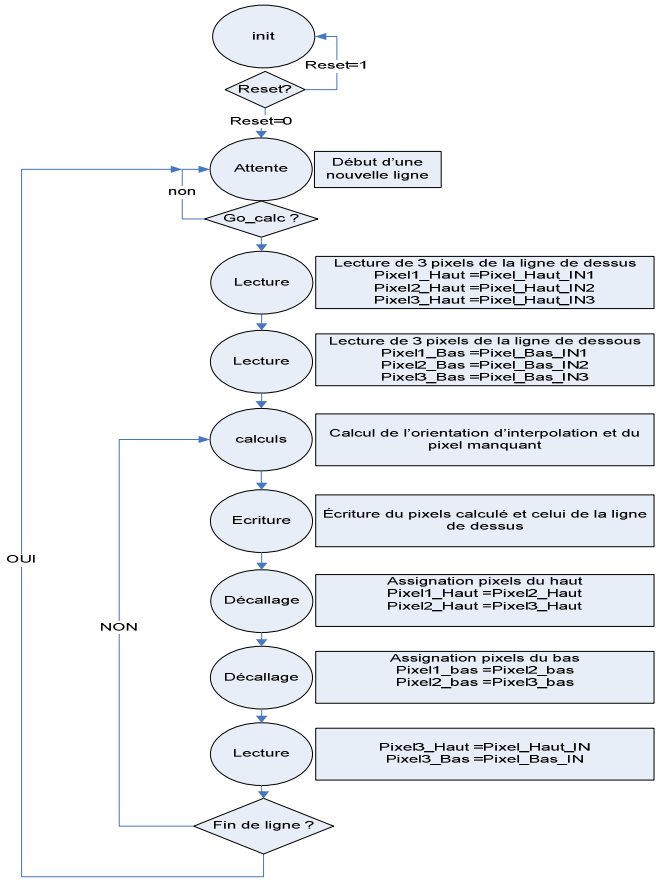


Figure 3.3 : Étapes du module de calcul

Un module de test a aussi été créé pour permettre de vérifier le bon fonctionnement du système. Une simulation est présentée à la figure 3.4. Nous avons placé comme donnée en entrée des valeurs de pixels qui s'incrémentent à chaque coup d'horloge de 13,5Mhz.

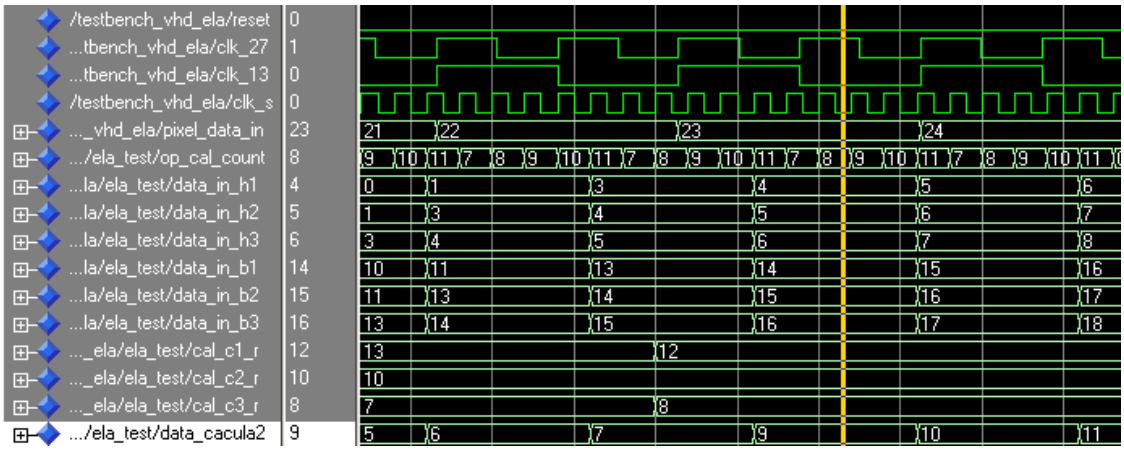


Figure 3.4 : Simulation du module de désentrelacement.

Pour des valeurs de pixels de la ligne de dessus de 4, 5, 6 et de 14, 15, 16 pour la ligne en dessous, on obtient une interpolation entre les valeurs 4 et 16 et une valeur de pixel manquant de 10.

3.4. Conclusion

En conclusion, le système implémenté a le comportement désiré mais la qualité de désentrelacement est étroitement reliée avec l'algorithme ELA. Pour obtenir une meilleure qualité d'image, il faudrait implémenter un autre module de traitement permettant de décrire des algorithmes plus complexes (par exemple la compensation de mouvement).

CHAPITRE 4. APPLICATION DES METHODOLOGIES PROPOSÉES À L'ALGORITHME ELA

4.1. Méthodologie ADL basée sur un processeur RISC

En suivant la méthodologie proposée nous avons nommé les différentes architectures correspondant aux étapes de cette dernière. La dénomination des ces différentes architectures est présenté au tableau 4.1.

Tableau 4.1 : Dénomination des architectures développées à partir d'un processeur RISC

RISC	Description
R0	Architecture de base
R1	Ajout de 3 fonctions spécialisées (average, abs, and min)
R2	Réduction du bus d'adresses 16 bits au lieu de 32 bits.
R3	Ajout d'une méga- instruction ELA qui prend 6 pixels et retourne le pixel manquant. Prototype : int ela (int, int ,int ,int, int ,int)
R4	Ajout de registres spécialisés et 3 nouvelles fonctions : elav2, shift_win et init_reg.
R4_clean	Retrait de toutes instructions inutilisées

L'application de la méthodologie générale de la figure 2.6 est détaillée ci-dessous.

A) Décomposition du code C en fonctions

La première étape de raffinage, nous conduit à regrouper tout d'abord à haut niveau le code source en petites fonctions :

int f_abs(int), int minimum(int,int,int) et int moyenne(int,int).

La première fonction calcule la valeur absolue, la deuxième trouve la direction d'interpolation et enfin le troisième calcule le pixel manquant en effectuant la moyenne de deux pixels dans la direction d'interpolation retenue.

B) Réduction du bus d'adresses de 32 à 16 bits

Cette opération nous a permi de réduire le nombre de cycles du traitement. En effet, le fait d'avoir les adresses sur 32 bits à l'encodage d'une instruction nous obligeait de prendre 2 cycles pour effectuer des accès en mémoire de données.

Une autre solution aurait été d'élargir la mémoire du programme à un nombre de bits supérieur à 32 pour pouvoir effectuer un accès en mémoire en un cycle aussi. Notez que cette opération n'aurait pas été possible avec d'autres types de méthodologies de conception qui imposent la largeur du bus d'adresses et de données.

C) Augmentation de la granularité

Cette manœuvre correspond au passage de la version RISC_R2 à RISC_R3 où l'on a créé une fonction : int ela (int, int ,int ,int, int ,int).

La boucle principale du code est présentée à la figure 4.1.

```
while ( i <  Nbre_champ_Pixel-1)  {
        ligne_r_ela[i-2]=ela(H1,H2,H3,B1,B2,B3);
        H1=ligne_dessus[i];
        B1=ligne_dessous[i];
        ligne_r_ela[i-1]=ela(H2,H3,H1,B2,B3,B1);
        H2=ligne_dessus[i+1];
        B2=ligne_dessous[i+1];
        ligne_r_ela[i]=ela(H3,H1,H2,B3,B1,B2);
        H3=ligne_dessus[i+2];
        B3=ligne_dessous[i+2];
        i=i+3;
   }
```

Figure 4.1 : Code C pour RISC_R3

Cette fonction (ELA) prend en paramètre 3 pixels du dessus et 3 pixels de la ligne de dessous. Elle détermine la direction d'interpolation et retourne la valeur du pixel manquant.

D) Ajout de registres spécialisés

Cette étape correspond au passage à la version RISC_R4. Nous avons crée trois fonctions supplémentaires : void int_reg(int, int), shiftwin(void) et enfin int ela(void). La boucle principale de traitement est présentée à la figure 4.2.

La première fonction (init_reg) nous permet d'initialiser les 8 registres spécialisés avec des valeurs contenues dans la mémoire de données. La deuxième fonction (shiftwin) nous permet de coulisser la fenêtre de traitement (Figure 3.2), nous permettant dans la boucle de traitement d'effectuer une seule lecture.

```
while ( i <  Nbre_champ_Pixel-2)
{
          ligne_r_ela[i]=elav2();  /// calcul du pixel manquant
          init_reg(3,ligne_dessus[i+2]); // lecture du nouveau pixel de la ligne de dessus
                                          //et initialisation du registre 3
          init_reg(7,ligne_dessous[i+2]); // lecture du nouveau pixel de la ligne
                                           // de dessous  et initialisation du registre 7
          shiftwin();   // décalage de la fenêtre de calcul
         i=i+1;
   }
```

Figure 4.2 : Code C pour RISC_R4

En effet, à chaque itération lorsqu'on calcule le pixel manquant, nous avons besoin de 6 pixels, 3 de la ligne de dessus et 3 de la ligne de dessous. Par la suite pour calculer le pixel suivant, quatre des six pixels ayant permis de calculer le pixel précédant sont réutilisés.

Pour finir, la dernière fonction elav2(), effectue tout le traitement à savoir la détermination de la direction d'interpolation puis, le calcul des pixels manquants. Au final, le code en C comprend ainsi la boucle principale de la figure 4.2.

E) Retrait des instructions inutilisées

Le nettoyage a été effectué manuellement en examinant le code assembleur généré par le compilateur. Toutes les instructions inutiles du jeu d'instructions ont été retirées de la description du processeur. Le tableau 4.2, représente la suppression de toutes les fonctions inutilisées et toutes les instructions spécialisées ajoutées au cours du processus d'exploration architecturale. Dans cette dernière architecture que nous avons nommé

R4_Clean, nous avons gardé seulement 18 instructions de base en plus des trois instructions spécialisées crées.

Tableau 4.2 : Instructions utilisées, A: arithmétique, Br: branchement, CMP: comparaison, S: spécialisées, LS: load/store, L: logique

instructions	add	addiu	andi	b	ba	bne	cmplt	elav2	Init
type	A	A	A	Br	Br	Br	CMP	S	S
Taux d'utilisation (%)	11.08	11.07	3.70	0.01	0.01	3.69	3.69	3.69	3.73
instructions	lbu	ldc	lw	nop	or	ori	sb	shiftwin	Sw
type	LS	LS	LS	A	L	L	LS	S	LS
Taux d'utilisation (%)	0.02	3.70	22.17	11.15	0.02	11.13	0.01	3.69	7.43

Cette étape permet de réduire le nombre de LUT de 62% tout en augmentant la fréquence maximale de 7% par rapport à l'étape précédente. Toutefois, le nombre de LUT est réduit de 27% par rapport au processeur original de base (RISC_R0). Au terme de cette méthodologie nous avons un facteur d'accélération de 11,33.

4.2. Méthodologie ADL basée sur un processeur VLIW

On atteint dans la première méthodologie proposée des limites d'accélération. Ces limites découlent du fait que l'on se concentre dans cette première étape sur la partie traitement en mettant moins d'emphase sur la partie communication.

De plus, en analysant le code assembleur généré de notre application, on constate qu'on retrouve plusieurs instructions de base successives sans dépendances de données qu'on pourrait ainsi exécuter au cours du même cycle d'horloge.

Pour palier à ces limites on propose de prendre comme point de départ de la seconde méthodologie un processeur VLIW ayant un bus de données de 128 bits.

A) Performances du processeur VLIW de base versus RISC

En prenant un processeur VLIW de base au lieu d'un processeur RISC, on parallélise les instructions de base quand la dépendance de données nous le permet. Nous présentons au tableau 4.3, les dénominations attribuées aux différents processeurs développés au cours de l'application de cette deuxième méthodologie.

Tableau 4.3 : Dénomination des architectures développées à partir d'un processeur VLIW

VLIW	**Description**
V0	Architecture de base
V1	Ajout de registres spécialisés et 3 nouvelles fonctions : elav2, shift_win et init_reg.
V2_asm	Retrait du compilateur et changement de modèle de programmation (Assembleur)
V3	- 256 bits de mémoire programme (possibilité d'exécuter 8 instructions de 32 bits en parallèle) - 192 bits de mémoire de données (8 pixels RGB de 24 bits)
V3_clean	Retrait de toutes instructions inutilisées
V_N	Extrapolation des résultats

Pour notre application nous obtenons un facteur d'accélération de 1,41 de la version VLIW_V0 comparativement à la version RISC_R0. L'explication de ce faible taux d'accélération provient du faible parallélisme de nos instructions de base. En effectuant

un profilage de la boucle principale de notre application nous obtenons les résultats de tableau 4.4.

Tableau 4.4 : Formation des syllabes

Nombre d'instructions dans une syllabe	Nombre de cycles d'exécution	Pourcentage d'utilisation (%)
1	84	56
2	58	38.7
3	6	4
4	2	1.3

Deux phénomènes interviennent ici pour expliquer ce faible facteur d'accélération comparativement à l'accélération totale possible:

- Dépendances de données dans l'algorithme : toutes les instructions ne peuvent pas être regroupées en groupe de 4 (formation de syllabes de 4 instructions)
- L'inefficacité du compilateur pour la gestion des résultats intermédiaires de calculs et l'utilisation de registres généraux.

B) Convergence des deux méthodologies

Cette étape propose de réutiliser les aboutissants des modifications successives de la méthodologie basée sur un processeur RISC. Nous modifions le VLIW de base avec les instructions et registres spécialisés optimisés au cours de la première méthodologie. En pratique, il s'agit de fusionner la version RISC_R4 et la version VLIW_V0 pour produire la version VLIW_V1.

Cette réunification se définit en trois grandes étapes :

- Insertion et adaptation des fonctions et registres spécialisés
- Modifications des règles du compilateur
- Ajustement de la taille du processeur aux données

L'adaptation des fonctions spécialisées se traduit par un changement mineur de syntaxe. Cette adaptation est illustrée à la figure 4.3 :

```
RESOURCE
{uint32 moy2_in1;    //  →  uint32 moy2_in1<0..3>;
 uint32 moy2_in2;    //  →  uint32 moy2_in2<0..3>;
 uint32 moy2_out; //  →     uint32 moy2_out<0..3>;
 uint32 Temp;        //  →     uint32 Temp<0..3>;
}
OPERATION moy2_ex IN pipe.DC {   / / →     OPERATION moy2_ex<id> IN pipe.DC
  […]
   Temp= moy2_in1 + moy2_in2;          / / →  Temp <id>=     moy2_in1<id> + moy2_in2 <id>;
    moy2_out = temp>>2;                / / →   moy2_out<id> = (temp<id> >>2);
  […]
}
```

Figure 4.3 : Adaptation des instructions RISC vers VLIW

Dans cet exemple, le paramètre <id> indique que l'on pourrait effectuer en parallèle 4 fonctions moyennes simultanément au cours du même cycle d'horloge au lieu de 4 cycles pour la version RISC_R4.

Après cette migration, en comparant avec le VLIW_V1 et le RISC_R4 on obtient une faible accélération de 1,07 : 20540 cycles comparativement à 22000 cycles.

On retombe dans la même problématique que la limite d'accélération de la méthodologie RISC car tout le potentiel d'exécution de processeur VLIW_V1 n'est pas exploité.

En restant à haut niveau dans la description de l'algorithme un fossé se creuse entre la puissance de calcul disponible et son utilisation. Nous devons nous attaquer à l'accélération de la dimension algorithmique du processus de conception.

C) Retrait du compilateur et passage en assembleur

Nous sommes confrontés à un choix: créer des fonctions contenant des instructions en assembleur avec le mot clef asm devant (Figure 4.4) ou tout simplement retirer complètement le compilateur C.

```
void main(void)
{
traitement();
}
asm traitement()
{
nop  |nop  | nop  | nop  |
nop  |nop  | nop  | nop  |
ori r11,r0,0x8004|ori r10,r0,0x82d4 | add r1,r0,1  | add r2,r0,2
}
```

Figure 4.4 : Retrait du compilateur

Le premier choix peut s'avérer très judicieux dans le cas où nous avons un code C très complexe et que le passage complet en assembleur serait très ardu, ou encore dans le cas où l'on voudrait uniquement accélérer certaines parties du code en C.

Dans notre cas, vue la faible complexité de notre application (ELA de base) nous avons décidé de retirer complètement le compilateur en C, et de passer du code C en code assembleur pour obtenir le maximum de performance. On obtient bien évidemment de meilleures performances qui sont rapportées à la ligne VLIW_V2asm (tableau 5.1).

Le passage du code C en assembleur nous permet d'avoir un gain de performance de 1,98 (20540/6883).

D) Ajout de N étages de traitement (ou élargissement du chemin des données) et ajout de fonctions spécialisées pour les « load » / « store »

N'ayant toujours pas atteint les objectifs désirés, une autre solution s'offre à nous : utiliser une architecture VLIW à N étages qui nous permet d'exécuter plus de 4 instructions en parallèles.

On propose une solution qui nous permet, avec une architecture VLIW et en élargissant le bus de données, d'exécuter plusieurs de ces traitements en parallèle.

On pourrait se dire intuitivement que de rajouter N étages permet d'avoir une accélération maximale d'un facteur de N. Ceci n'est vrai uniquement que si lors de l'ajout de ces N étages on ne crée pas de fonctions spécialisées qui tirent avantage de l'élargissement de la mémoire de programme et du bus de données. Dans notre cas, nous avons créé, lors du passage de VLIW_V2_asm à VLIW_V3, quatre fonctions spécialisées :

- Une fonction *load* qui charge en mémoire 192 bits (soit 8 pixels RGB encodés sur 24 bits)
- Une fonction *split* qui initialise chacun des registres spécialisés
- Une fonction *concatenation* qui récupère les résultats de chacun des calculs ela (dans des registres spécialisés) et les placent dans un registre de 192 bits

- Et enfin une fonction *store* qui sauvegarde en mémoire 192 bits (soit 8 pixels RGB encodés sur 24 bits)

```
nop | nop|  nop| nop| nop | nop|  nop|  nop
ori r11,r0,0x1E|ori r10,r0,0x00 | ori r12,r0, 0x40 | add r1,r0,1  | add r2,r0,2  | add r4,r0, 720   |add r8,r0, 8   |
nop   ; Initialisation des registres de sauvegarde et de lecture
_boucle1:
load(r11,r0) | nop|  nop| nop | nop | nop| nop| nop     ; charge 8 pixels de la ligne de dessus
load(r10,r1) | nop|  nop| nop | nop | nop| nop| nop     ; charge 8 pixels de la ligne de dessous
split() | nop|  nop| nop | nop | nop| nop| nop              ; repartie chaque pixels dans les registres specialisé
sub r4,r4,r8 |add r11,r11,r1|add r10,r10,r1|add r12,r12,r1| nop | nop|  nop| nop   ; Décrémentation du
                                        ; compteur et mis à  jour des registres de sauvegarde et de lecture
elav2() |elav2() |elav2()  | elav2() |elav2() |elav2()  | elav2() |elav2()  ; évaluation des fonctions ELA
concatenation() | nop|  nop| nop | nop | nop| nop| nop     ; Concaténation des résultats dans un registre de
24 bits
store(r12,r0) | nop|  nop| nop | nop | nop| nop| nop  ; Sauvegarde des résultats
bne r4,_boucle1 | nop | nop| nop | nop | nop | nop| nop   ; saut au début de la boucle si la ligne n'est pas fini
```

Figure 4.5 : Code assembleur pour la version VLIW_V3

On remarque dans la description de l'algorithme que toutes les unités du pipeline exécutent une instruction à chaque cycle.

À cette étape la flexibilité et la description haut niveau qu'offrait le code C est complètement perdue; il faut s'assurer que d'une instruction à l'autre il n'y ait pas de dépendances de données.

E) Nettoyage du processeur

En enlevant le compilateur et en étant à très bas niveau, on a l'avantage à la fin de notre processus d'exploration d'enlever toutes les instructions inutilisées. Cette opération

apparemment anodine est une grande force de l'utilisation de notre environnement de travail. Dans des méthodologies d'ajout d'instructions spécialisées cette opération n'est souvent pas possible car nous sommes souvent confrontés à un noyau dur d'instructions de base nécessaires pour le fonctionnement du compilateur. C'est le cas dans les méthodologies de conception avec Xtensa de la société Tensilica par exemple.

F) Vers un nombre d'étages d'exécution maximal

On pourrait extrapoler les résultats pour connaitre dans notre application actuelle quelle serait l'accélération pour un nombre de cycles minimum impliquant ainsi un nombre d'étages de traitement maximum. Nous aurions ainsi un code assembleur qui se présenterait comme l'illustre la figure 4.6 :

```
ori r11,r0,0x1E | ori r12,r0, 0x40 | nop  | nop  | nop  | nop | nop …….
        ; Initialisation des adresses des deux lignes
load(r11,r0) | nop|  nop| nop | nop | nop| nop| nop  | nop | nop ……..
        ; Chargement des 720 pixels de la ligne de dessus
load(r10,r1) | nop|  nop| nop | nop | nop| nop| nop | nop | nop  ……..
        ; Chargement des 720 pixels de la ligne de dessous
split() | nop|  nop| nop | nop | nop| nop| nop  | nop | nop  | nop  ……..
        ; Initialisation des registres spécialisés de chaque étage
ela() |ela() |ela()  | ela() |ela() |ela()  | ela() |ela()|ela()|ela()|ela() ……
        ; Calcul de chacun de des pixels
concatenation() | nop|  nop| nop | nop | nop| nop| nop
        ; Récupération de chaque résultat de chaque étage
 store (r12,r0) | nop|  nop| nop | nop | nop| nop| nop
        ; Sauvegarde  des 720 pixels de la ligne manquante
```

Figure 4.6 : Syntaxe en LISA 2.0 permettant l'ajout de N étages de traitements

Ainsi au total on aurait exactement 7 cycles d'exécution pour compléter une ligne de 720 pixels manquants. D'un point de vu matériel ceci se traduit par N unités de calcul en parallèles représentés à la figure 4.7.

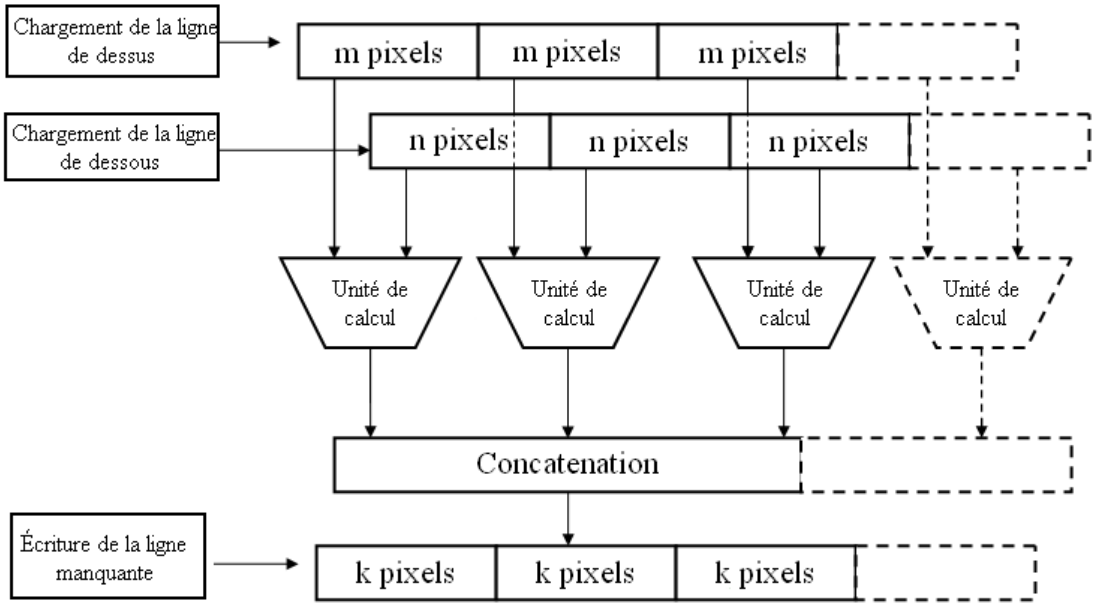


Figure 4.7 : Ajout de N étages de traitements

4.3. Conclusion

Dans ce chapitre nous avons appliqué nos méthodologies à l'algorithme ELA. La première méthodologie se caractérise par la création de fonctions de plus en plus complexes et par une mise à l'échelle de notre processeur. Le but de cette première méthodologie est d'obtenir un processeur plus performant par ajout d'instructions spécialisées. La mise à l'échelle du processeur peut être perçue comme facultative, elle permet tout simplement d'avoir dès le départ de notre méthodologie un processeur dédié

pour notre type de données. La deuxième méthodologie quant à elle reprend tous les efforts obtenus avec la première, mais en se basant sur une architecture plus performante. Au cours des différentes étapes, cette méthodologie se distingue de la première car en plus d'ajouter des instructions spécialisées, nous modifions intégralement l'architecture de base. Le but ultime de cette méthodologie est d'obtenir des taux d'accélération plus important en s'attaquant aux limitations de la première.

CHAPITRE 5. RESULTATS ET DISCUSSION

Nous présentons au tableau 5.1 les résultats de synthèse ainsi que les performances des différents processeurs développés lors de l'application de nos deux méthodologies à un algorithme ELA à trois pixels.

Tableau 5.1 : Résultats de synthèse

Dénomination de l'architecture développée	Surface normalisée pour le FPGA Virtex 2 Pro (2vp30ff896-7)			Fréquence maximale (MHz)	Ligne de code (~complexité)	Nombre de cycles pour compléter une ligne	Temps de calcule d'une image complète 720 × 480 pixel image (ms)	Accélération	Facteur A.T normalisé
	Nombre de Tranches (normalisé)	Nombre de Flip Flops (normalisé)	Nombre de LUTs (normalisé)						
RISC_R0	1.00 (4748)	1.00 (1027)	1.00 (9199)	83.80	80	260 K	745.00	1.00	1
RISC_R1	1.55	1.66	1.51	84.70	35	235 K	667.00	1.11	13.55
RISC_R2	1.47	1.33	1.45	83.40	35	180 K	518.00	1.44	9.78
RISC_R3	1.76	1.66	1.72	82.40	22	22.3 K	65.10	11.44	1.44
RISC_R4	2.02	1.66	1.96	75.20	24	22 K	70.20	10.61	1.78
RISC_R4_clean	0.76	1.66	0.72	80.40	24	22 K	65.70	11.33	0.69
VLIW_V0	2.64	2.33	2.58	72.15	80	184 K	611.42	1.22	20.85
VLIW_V1	3.71	3.00	3.60	62.23	24	21 K	77.34	9.63	3.67
VLIW_V2_asm	3..38	2.66	3.27	58.62	30	6.9 K	28.18	26.60	1.20
VLIW_V3	6.82	6.00	6.45	64.07	12	820	3.07	242.00	0.26
VLIW_V3_clean	2.64	3.66	2.27	74.30	12	820	2.64	282.00	0.085
VLIW_N_stages	*238*	*330*	*204*	*74.30*	*7*	*7*	*22.50 ns*	*33111.00*	*0.065*

On retrouve sur chaque ligne de ce tableau les différents processeurs que nous avons conçu et sur chaque colonne correspondante les performances associées. Le premier

critère de performance, la surface occupée sur un FPGA Virtex 2 pro (2vp30), a été subdivisé en trois colonnes : le nombre de Tranches ou « *slices* » , le nombre de flip flop et enfin le nombre de LUT (Lookup table). Il est à noter de que le maximum du nombre de Tranches, Flip Flop et LUTs sont respectivement 13696, 27392 et 27392. Ainsi si on prend comme exemple la version RISCO_R0 nous avons une surface occupée de 34% de Tranches, 3% de Flip Flop et 33% de LUTs.

Pour une meilleure lecture, nous avons normalisé ces valeurs par rapport à l'architecture RISC_R0 pour faire ressortir l'évolution de la surface occupée des processeurs au cours des étapes successives des méthodologies. La cinquième colonne présente les fréquences maximales d'exécution des processeurs. Cette fréquence est reliée à la technologie utilisée. Les autres colonnes font ressortir les performances de nos processeurs lorsqu'ils sont utilisés pour exécuter l'algorithme ELA à trois pixels.

La dernière colonne présente le facteur AT (Area Time) qui est le produit de la surface occupée par le temps d'exécution du processeur pour désentrelacer une image de définition 720x480. C'est notre critère ultime de comparaison des architectures, car il fait intervenir sur le même pied d'égalité la surface, la fréquence maximale des processeurs et le nombre de cycles total d'exécution de l'algorithme. Nous avons aussi ajouté à la dernière ligne en italique l'extrapolation des résultats qui découleraient de l'étape F de la section 4.2.

Pour la méthodologie de conception basée sur un processeur RISC, on remarque que pour un gain d'un facteur de 11 en accélération, on a diminué notre facteur de performance d'environ 30% (facteur A.T).

Les résultats de conception obtenus pour l'architecture RISC sont réutilisés dans l'architecture VLIW. En effet, la version VLIW_V1 reprend les modifications effectuées dans RISC_R4. La dernière ligne du tableau, qui présente une extrapolation de résultats, nous indique qu'en ajoutant le nombre d'étages maximum possible, on arriverait à une faible amélioration du niveau de performance, à en juger par le facteur AT.

De plus, pour calculer le temps de traitement, on a supposé que la fréquence d'horloge reste constante alors que lorsqu'on double le nombre d'étages, on s'aperçoit qu'il y a une diminution de la fréquence maximale d'environ 11,2%. Il faut remarquer ici que les étages d'exécution que l'on rajoute sont en parallèle, et l'ajout de N étages diminuera principalement la fréquence d'horloge à cause du chemin critique qui se trouve dans le module de décodage d'instructions. Le tableau 5.2: présente les correspondances entre les étapes de nos deux méthodologies.

Tableau 5.2 : Comparaison entre l'exploration architecturale à partir d'un processeur RISC et celle de VLIW

Évolution	**RISC**	**VLIW**	**Description**
0	R0	V0	Architecture de base
1	R1	-	Ajout de 3 fonctions spécialisées ('average', 'abs', et 'min')
2	R2	-	Réduction du bus d'adresses 16 bits au lieu de 32 bits.
3	R3	-	Ajout d'une méga- instruction ela qui prend 6 pixels et retourne le pixel manquant. Prototype : int ela (int, int, int, int, int, int)
4	R4	V1	Ajout de registres spécialisés et 3 nouvelles

			fonctions : elav2, shift_win et init_reg.
5	-	V2_asm	Retrait du compilateur et changement de modèle de programmation (Assembleur)
6	-	V3	- 256 bits de mémoire programme (possibilité d'exécuter 8 instructions de 32 bits en parallèle) - 192 bits de mémoire de données (8 pixels RGB de 24 bits)
7	R4_clean	V3_clean	Retrait de toutes instructions inutilisées
8		V_N	Extrapolation des résultats

Le tableau 5.2 permet de présenter nos deux explorations architecturales comme deux méthodes distinctes et non successives. Il en souligne aussi les points communs. Les différentes évolutions se caractérisent par les types de modifications effectuées. La deuxième colonne présente les processeurs résultants de la première méthodologie tandis que la troisième colonne présente ceux résultants de la seconde méthode. Pour finir, la dernière colonne présente les caractéristiques ou modifications effectuées pour chacun des processeurs.

Si on nous propose un nouvel algorithme, il serait possible de sauter la première méthodologie et de passer directement à la seconde. Les différentes étapes consistent a effectuer les modifications des évolutions 1 à 3 pour le processeur VLIW. Néanmoins cette approche n'est pas conseillée, car cela reviendrait à sauter toute l'étape d'optimisation des instructions critiques de notre algorithme de départ avec un processeur de faible complexité. Un autre point important à considérer est que les ressources utilisées par les deux méthodes peuvent être, dépendamment des applications,

une contrainte importante et cruciale. Ainsi l'option de passer directement à la deuxième méthode devrait être soigneusement étudiée car cela équivaut à doubler la complexité du processeur de base.

Les figures 5.1 et 5.2 représentent les performances des différents processeurs en fonction du temps de production d'une image et le facteur A.T. respectivement. L'axe des abscisses, que l'on a appelé évolution, correspond à un même type de manipulations effectuées dans les deux différentes méthodologies. On retrouve en détail la description de ces évolutions au tableau 5.2.

En comparant les deux méthodes à partir des figures 5.1 et 5.2, on remarque que la version RISC est plus efficace en terme de produit AT pour les évolutions de 0 à 4. Le faible taux d'instructions parallélisées automatiquement par le compilateur ainsi que l'augmentation par un facteur de 4 de la surface explique ce constat.

Le retrait du compilateur ainsi que l'ajout d'étages d'exécution en parallèle pour la version VLIW permet par la suite d'aller chercher un facteur d'accélération de 282.

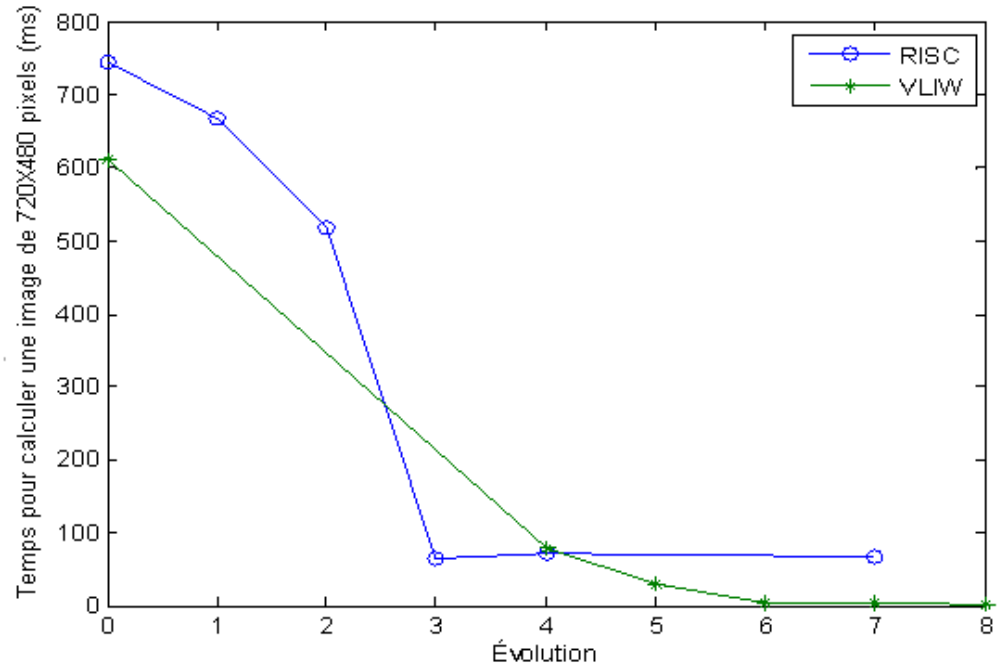


Figure 5.1 : Évolution du temps de production d'une image au cours de l'exploration architecturale

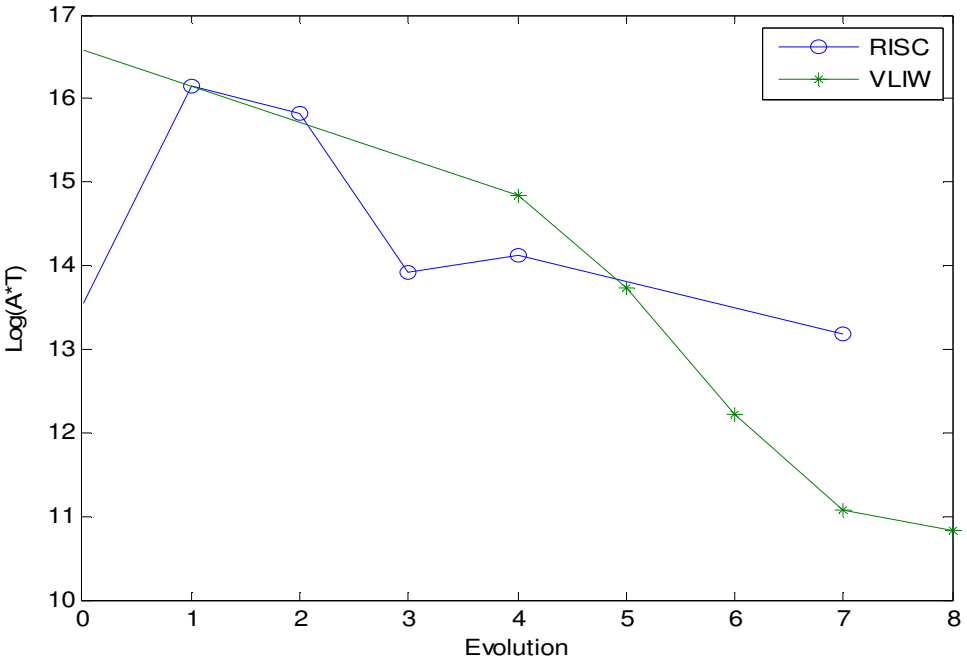


Figure 5.2 : Évolution du facteur A.T au cours de l'exploration architecturale

La figure 5.3 reprend nos résultats en nous offrant une vision multidimensionnelle de notre travail. Elle a pour avantage de donner un aperçu des impactes synergiques des deux méthodologies.

L'axe « évolution RISC » de la figure 5.3 regroupe des actions qui consistent à ajouter des registres et fonctions spécialisés. L'axe VLIW représente une parallélisation de ces instructions spécialisées, tandis que l'axe SIMD représente la parallélisation de plusieurs traitements de même type.

On pourrait ajouter à ce graphique une ou plusieurs autres dimensions représentant une catégorisation des efforts de conception visés. Par exemple, on pourrait ajouter un axe pour la consommation de puissance et un autre pour différents langages de description utilisés pour coder notre algorithme (assembleur, C, C++, System C).

Dans le cas ou l'on chercherait à améliorer la consommation de puissance on pourrait ainsi à partir de la version V3_clean effectuer des modifications à ce processeur dans le but de réduire sa consommation de puissance. Ceci rajouterait donc une autre dimension à la figure 5.3 qui représenteraient les accélérations obtenues avec différentes évolutions du processeur V3_clean dans le but de réduire la puissance consommée.

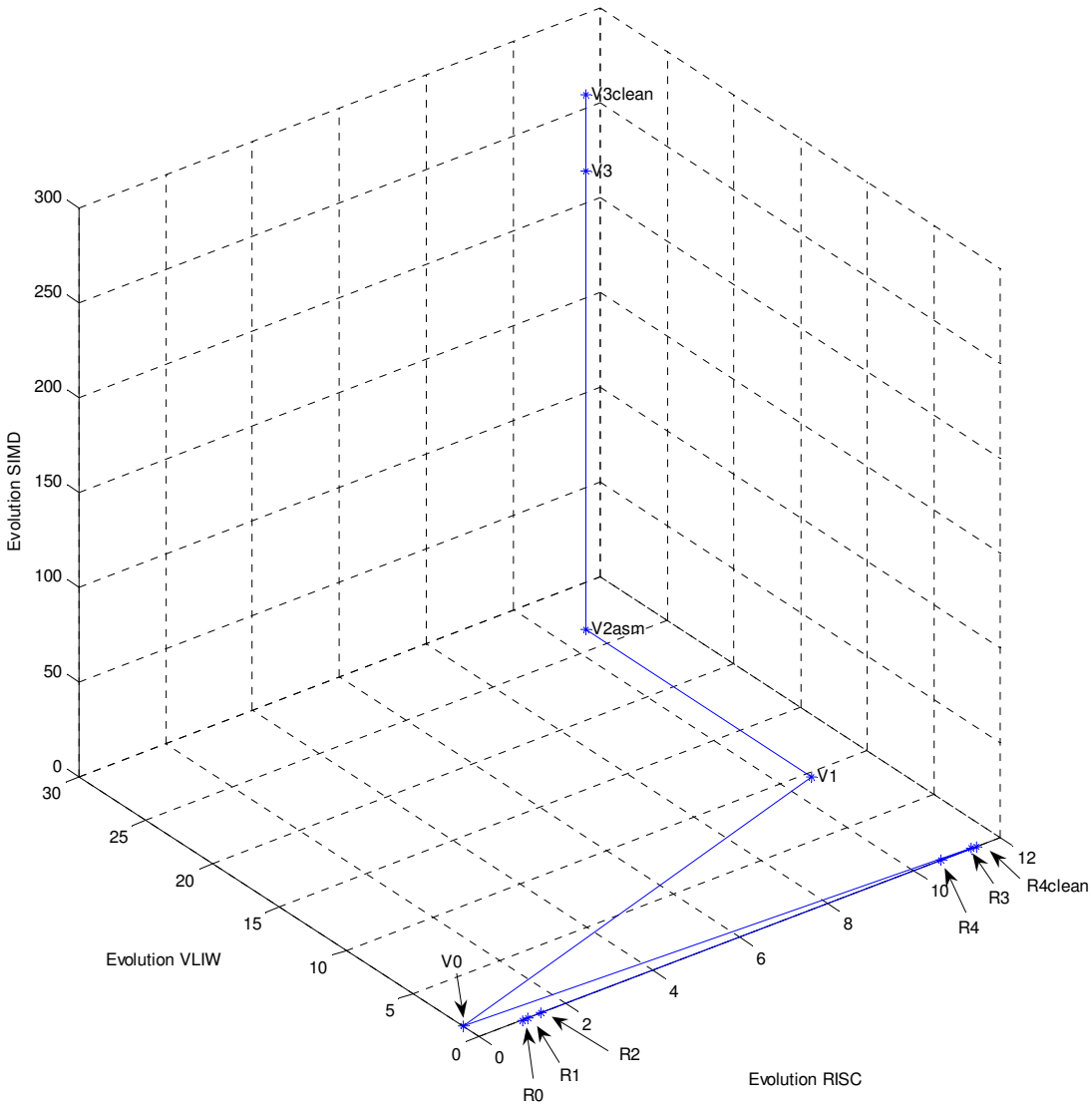


Figure 5.3 : Résumé des dimensions d'accélération explorées

Nous présentons à la figure 5.4 l'évolution des instructions au cours en appliquant nos deux méthodologies.

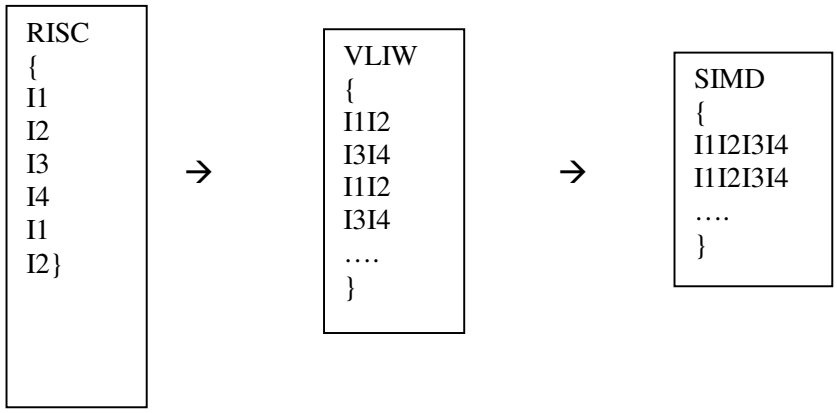


Figure 5.4 : Apport synergique des différentes dimensions d'accélération

La première colonne représente le résultat des modifications de la première méthodologie. Nous avons à chaque ligne l'exécution d'une instruction qui idéalement a été spécialisée pour notre algorithme.

Dans la deuxième colonne nous tentons de paralléliser ces instructions dans le but d'exécuter à chaque cycle au moins deux instructions. Cette parallélisation n'est possible que dans le cas où nos instructions résultantes de la première méthodologie sont indépendantes. À chaque cycle d'exécution nous avons aussi une évaluation des différentes instructions. Pour finir nous essayons de regrouper tous les différents traitements en un même traitement. Cette manœuvre a pour but d'exécuter à chaque cycle un même traitement.

On peut remarquer qu'effectuer ce type de manipulation a pour effet d'augmenter la surface occupée par le processeur tout en conservant le chemin critique constant. En

effet en supposant que nos instructions soient parallélisables, le chemin critique ne sera augmenté que par la partie décodage des instructions. Cette parallélisation illustrée à la figure 4.7 : est reporté dans le tableau 5.1 de synthèse qui confirme nos propos.

D'un point de vue global, toutes les étapes de conception ont été passées en revue. Nous sommes partis d'une implémentation du système global comprenant un système d'acquisition, un module de traitement et un système d'affichage. L'aspect modulaire d'un système de traitement et l'exploration architecturale effectuée permettent d'entrevoir une migration possible vers d'autres types de traitements plus complexes tels que la compensation de mouvement.

La deuxième partie du travail consiste à concevoir le processeur à proprement dit. Nous sommes partis d'une implémentation purement matérielle (VHDL) à la réalisation d'un processeur dédié pour le même type de traitement de base (ELA).

Deux méthodologies de conception avec un langage ADL ont été proposées. La première méthodologie a comme particularité fondamentale d'optimiser les instructions spécialisées tout en supportant une description de l'algorithme en C.

Après avoir réalisé que l'accélération d'exécution des instructions atteint, après plusieurs itérations, une valeur maximale avec la première méthodologie, nous avons développé la deuxième. Cette deuxième méthode reprend toutes les améliorations obtenues à l'aide de la première, tout en intégrant la notion de parallélisation des instructions.

CONCLUSION

Cette recherche a exploré différentes méthodes de conception applicables à la conception de circuits et processeurs spécialisés. Nous avons notamment considéré les processeurs d'usage général, les ASIP et les ASIC. Il s'en dégage plusieurs méthodologies de conception qui, selon les besoins et les ressources disponibles en temps et en coûts, sont plus ou moins avantageuses.

Nos besoins étant de concevoir des processeurs pour des traitements vidéo nous avons mis une emphase sur les défis que l'on rencontre dans ce type d'application. Nous avons passé en revue les défis de conception en traitement vidéo qui motive notre travail sur la conception de processeurs dédiés. Parmi ces traitements, le désentrelacement a été retenu comme algorithme de base pour développer nos processeurs. Ce traitement, omniprésent dans le marché multimédia, permet de reconstituer des séquences vidéo complètes à partir des séquences vidéo ayant une moitié moins d'informations (vidéo entrelacée).

La conception de processeurs dédiés a été par la suite explorée et choisie pour répondre à nos besoins; plus particulièrement, en empruntant une voie de conception de processeur axée sur un ADL. Parmi ces langages de conception, nous avons choisi LISA 2.0 qui nous offrait au début de nos travaux les performances requises pour atteindre de grands niveaux d'accélération. En parallèle avec ces explorations théoriques et algorithmiques, nous avons développé un environnement de test qui répond à une des problématiques de départ à savoir comment concevoir un environnement de test

complet, permettant de visualiser les résultats de l'implémentation matérielle. L'aspect modulaire de notre système, développé sur une plaquette de développement de Xilinx, nous a permis d'implémenter et de tester au fur et à mesure nos différents processeurs vidéo dans un système complet. Ce système comprend un lecteur de DVD fournissant des signaux vidéo entrelacés, des modules de traitements permettant d'effectuer le désentrelacement, et finalement un écran VGA pour afficher les résultats.

Une fois s'être assurés de la modularité de notre système, nous nous sommes concentrés sur le processeur proprement dit. Nous avons développé des méthodologies de conception originales et itératives basées sur un ADL. Ces méthodes indiquent des étapes structurées et itératives afin d'obtenir des accélérations d'algorithmes. L'aspect générique de ces méthodes a été un souci constant dans l'élaboration de ces dernières.

En nous basant sur l'algorithme ELA, nous avons obtenu un graphique 3D récapitulatif qui nous a montré l'effet des différentes actions apportées au cours de nos travaux en prenant comme point de départ des processeurs de base RISC et VLIW.

Dans un premier temps nous avions choisi le processeur RISC pour une entrée en matière avec le langage LISA 2.0, mais aussi pour sa faible complexité. Nous avons raffiné les instructions d'un processeur de base et créé des instructions spécialisées pour arriver finalement a des plafonds d'accélération de l'ordre de 11 avec un facteur de performance AT de 0,69. Ayant atteint des plafonds d'accélération, nous avons migré nos efforts de conception vers un processeur VLIW. Cette migration s'est accompagnée d'une part, par la réutilisation complète des instructions déjà optimisées et d'autre part par la création de macro instructions regroupant plusieurs instructions raffinées

identiques (SIMD). Nous obtenons en bout de ligne une accélération de 282 avec un facteur de performance AT de 0,085.

Les objectifs initiaux du projet ont été atteints avec succès. Nous avons implémenté des processeurs vidéo pour réaliser des algorithmes de désentrelacement vidéo en temps réel. Ces derniers ont été testés dans un environnement de test complet, permettant de visualiser les résultats de l'implémentation matérielle. Enfin nous avons défini, d'une part, une méthode de conception de matériel basée sur une description en VHDL et d'autre part, deux méthodes de conception basée sur un ADL permettant l'accélération d'algorithmes de traitement vidéo.

Travaux futurs

Sur le plan de l'architecture du système, il faudrait développer une architecture qui comprend un bus de données pour inclure une RAM externe. L'idéal aurait été de développer du code pour sauvegarder et lire des trames complètes en vue d'utiliser des algorithmes plus complexes comme la compensation de mouvement. Cette nécessité découle de la revue de littérature, où l'on met en évidence que les meilleurs algorithmes de désentrelacement sont des algorithmes inter-trames qui nécessitent de conserver en mémoire plusieurs trames.

Du côté de la méthodologie de conception basée sur du ADL, les programmes réalisés devraient dans le même ordre d'idées inclure des transactions de bus. La création de bus de données est supportée par les outils de Coware et devrait être explorée.

Un autre point important est que les résultats de la deuxième méthodologie de conception devraient être repris dans le cadre d'un article de la même façon que la

première méthodologie présentée à la conférence Newcas [23]. Le manque de temps a malheureusement rendu impossible l'écriture de ce second article.

RÉFÉRENCES

[1] Intel : " Intel high-performance consumer desktop microprocessor timeline", [En ligne], disponible : http://www.intel.com/pressroom/kits/core2duo/pdf/microprocessor_timeline.pdf

[2] Intel : " Intel's First Microprocessor the Intel 4004 ", [En ligne], disponible : http://www.intel.com/museum/archives/4004.htm

[3] Intel : " Core 2 Duo Processors", [En ligne], disponible : http://www.intel.com/products/processor/core2duo/index.htm

[4] C. T. Johnston, K. T. Gribbon, D. G. Bailey, "Implementing Image Processing Algorithms on FPGAs", *Proc. of the 11th Electronics New Zealand Conference*, pp.118-123, November 2004.

[5] Callico G., Lopez S., Sosa O., Lopez J.F., Sarmiento R., "Analysis of fast block matching motion estimation algorithms for video super-resolution systems," *Consumer Electronics, IEEE Transactions on* , vol.54, no.3, pp.1430-1438, August 2008

[6] Oklobdzija, V.G., "Architecture for single-chip ASIC processor with integrated floating point unit," *System Sciences, 1988. Vol.I. Architecture Track, Proceedings of the Twenty-First Annual Hawaii International Conference on* , vol.1, no., pp.221-229, 5-8 Jan 1988

[7] Kim, S.D., Lee, J.H, Yang, J.M., Sunwoo, M.H., and Oh, S.K., "Novel instructions and their hardware architecture for video signal processing," in Proc. *IEEEInt. Symp. Circuits Syst.,* May 2005.

[8] Jae S. Lee, Young S. Jeon, and Myung H. Sunwoo, “Design of new DSP instructions and their hardware architecture for high-speed FFT,” in Proc. IEEE Workshop on Signal Processing Syst., Sept. 2001, pp. 80-90.

[9] Wang, Y.; Tang, Y.; Jiang, Y.; Chung, Y.-G.; Song, S.-S.; Lim, M.-S., "Novel Memory Reference Reduction Methods for FFT Implementations on DSP Processors," *Signal Processing, IEEE Transactions on [see also Acoustics, Speech, and Signal Processing, IEEE Transactions on]* , vol.55, no.5, pp.2338-2349, May 2007

[10] Ricardo E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, vol. 20, no. 2, pp. 60-70, Mar/Apr 2000.

[11] Beucher, N.; Belanger, N.; Savaria, Y.; Bois, G., "Motion Compensated Frame Rate Conversion Using a Specialized Instruction Set Processor," *Signal Processing Systems Design and Implementation, 2006. SIPS '06. IEEE Workshop on* , vol., no., pp.130-135, Oct. 2006

[12] Meehan, Joseph; Yoo, Youngjun Francis; Ching-Yu Hung,; Polley, Mike, "Media processor architecture for video and imaging on camera phones," *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on* , vol., no., pp.5340-5343, March 31 2008-April 4 2008

[13] Kim, Sung D.; Hyun, Choong J.; Sunwoo, Myung H., "VSIP : Implementation of Video Specific Instruction-set Processor," *Circuits and Systems, 2006. APCCAS 2006. IEEE Asia Pacific Conference on* , vol., no., pp.1075-1078, 4-7 Dec. 2006

[14] Saponara, S.; Fanucci, L.; Marsi, S.; Ramponi, M.; Kammler, D.; Witte, E. M.,"Application-Specific Instruction-Set Processor for Retinex-Like Image and Video Processing," IEEE Transactions on Circuits and Systems II: Express Briefs, vol.54, no.7, pp. 596-600, July 2007.

[15] Mishra, P.; Dutt, N., "Architecture description languages for programmable embedded systems," Computers and Digital Techniques, IEE Proceedings - , vol.152, no.3, pp. 285-297, 6 May 2005

[16] Muller, O.; Baghdadi, A.; Jezequel, M., "From Application to ASIP-based FPGA Prototype: a Case Study on Turbo Decoding," Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium on , vol., no., pp.128-134, 2-5 June 2008

[17] Rashid, M.; Apvrille, L.; Pacalet, R., "Application Specific Processors for Multimedia Applications," Computational Science and Engineering, 2008. CSE '08. 11th IEEE International Conference on , vol., no., pp.109-116, 16-18 July 2008

[18] Hoffmann, A.; Fiedler, F.; Nohl, A.; Parupalli, S., "A methodology and tooling enabling application specific processor design," VLSI Design, 2005. 18th International Conference on , vol., no., pp. 399-404, 3-7 Jan. 2005

[19] Hossein Mahvash Mohammadi; Yvon Savaria; J.M. Pierre Langlois, "Real Time ELA De-Interlacing with the Xtensa Reconfigurable Processor," IEEE North-East Workshop on Circuits and Systems, pp.25-28, June 2006.

[20] G. De Haan and E. B. Bellers, “Deinterlacing – An Overview,” Proceedings of the IEEE, vol. 86, no. 9, pp.1839 – 185 Sept. 1998.

[21] De Haan, G.; Bellers, E.B., "Deinterlacing-an overview," Proceedings of the IEEE , vol.86, no.9, pp.1839-1857, Sep 1998

[22] T. Chen, H.R. Wu, and Z.H. Yu, “An efficient edge line average interpolation algorithm for deinterlacing,” Proc. SPIE: Visual Cmmunications and Image Processing, vol. 4067, pp. 1551-1558, 2000

[23] Ngoyi, G.-A.B.; Pierre Langlois, J.M.; Savaria, Y., "Iterative design method for video processors based on an architecture design language and its application to ELA deinterlacing," Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on , vol., no., pp.37-40, 22-25 June 2008

[24] Hongbin Sun; Nanning Zheng; Chenyang Ge; Dong Wang; Pengju Ren, "An Efficient Motion Adaptive De-interlacing and Its VLSI Architecture Design," Symposium on VLSI, 2008. ISVLSI '08. IEEE Computer Society Annual , vol., no., pp.455-458, 7-9 April 2008

[25] Yanfei Shen; Dongming Zhang; Yongdong Zhang; Jintao Li, "Motion adaptive deinterlacing of video data with texture detection," Consumer Electronics, IEEE Transactions on , vol.52, no.4, pp.1403-1408, Nov. 2006

[26] E.B. Bellers and G. de Haan Proc., Advanced de-interlacing techniques ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing, Mierlo, The Netherlands, November 1996, pp. 1-13(3-a)

[27] Shyh-Feng Lin, Yu-Lin Chang, and Liang-Gee Chen, Motion adaptive de-interlacing by horizontal motion detection and enhanced ELA processing, ISCAS 2003, pp. II696-II699 (1-c)

[28] Tero Koivunen, "Motion detection of an interlaced video signal", IEEE Transactions on Consumer Electronics, Vol. 40, No. 3, AUGUST 1994, pp. 753-760

[29] Dongil Han, Chang-Yong Shin, Seung-Jong Choi, and Jong-Seok Park, "Motion adaptative 3-D de-interlaced algorithm based on brightness profile pattern difference, IEEE Transactions on Consumer Electronics, Vol. 45, No. 3, AUGUST 1999 , pp.690-697 (1-c)

[30] Gerard de Haan and Rogier Lodder, "De-interlacing of video data using motion vectors and edge information " Digest of the ICCE'02, Jun. 2002, pp. 70-71, (1-d)

[31] Zhu and K. K. Ma, “A new diamond search algorithm for fast block matching motion estimation,” Proc. of Int. Conf. Information, Communications and Signal Processing, vol. 1, pp. 292-6, 1997.

[32] Digilent : ”Design Resources for Digital Engineers ” [En ligne], disponible : www.digilentinc.com

[33] P. Ienne and R. Leupers, “From prêt-à-porter to tailor-made” in Customizable Embedded Processors, Morgan-Kaufman, 2007.

[34] Wei Qin; Ben-Tzur, A.; Gutkovich, B., "An ADL for Functional Specification of IA32," Microprocessor Test and Verification, 2007. MTV '07. Eighth International Workshop on , vol., no., pp.119-127, 5-6 Dec. 2007

[35] M. R. Barbacci. Instruction set processor specifications (ISPS): The notation and its applications. IEEE Transactions on Computers, C-30(1):24–40, 1 1981.

[36] A. Fauth, J. V. Praet, and M. Freericks. Describing instruction set processors using nML. In Proceedings of Conference on Design Automation and Test in Europe, pages 503–507, Paris, France, 1995.

[37] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In Proceedings of Design Automation Conference, pages 299–302, June 1997.

[38] G. Zimmerman. The MIMOLA design system: A computeraided processor design method. In Proceedings of Design Automation Conference, pages 53–58, June 1979.

[39] H. Akaboshi. A Study on Design Support for Computer Architecture Design. PhD thesis, Department of Information Systems, Kyushu University, Japan, 1996.

[40] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA –machine description language for cycle-accurate models of programmable DSP architectures. In Proceedings of Design Automation Conference, pages 933–938, 1999.

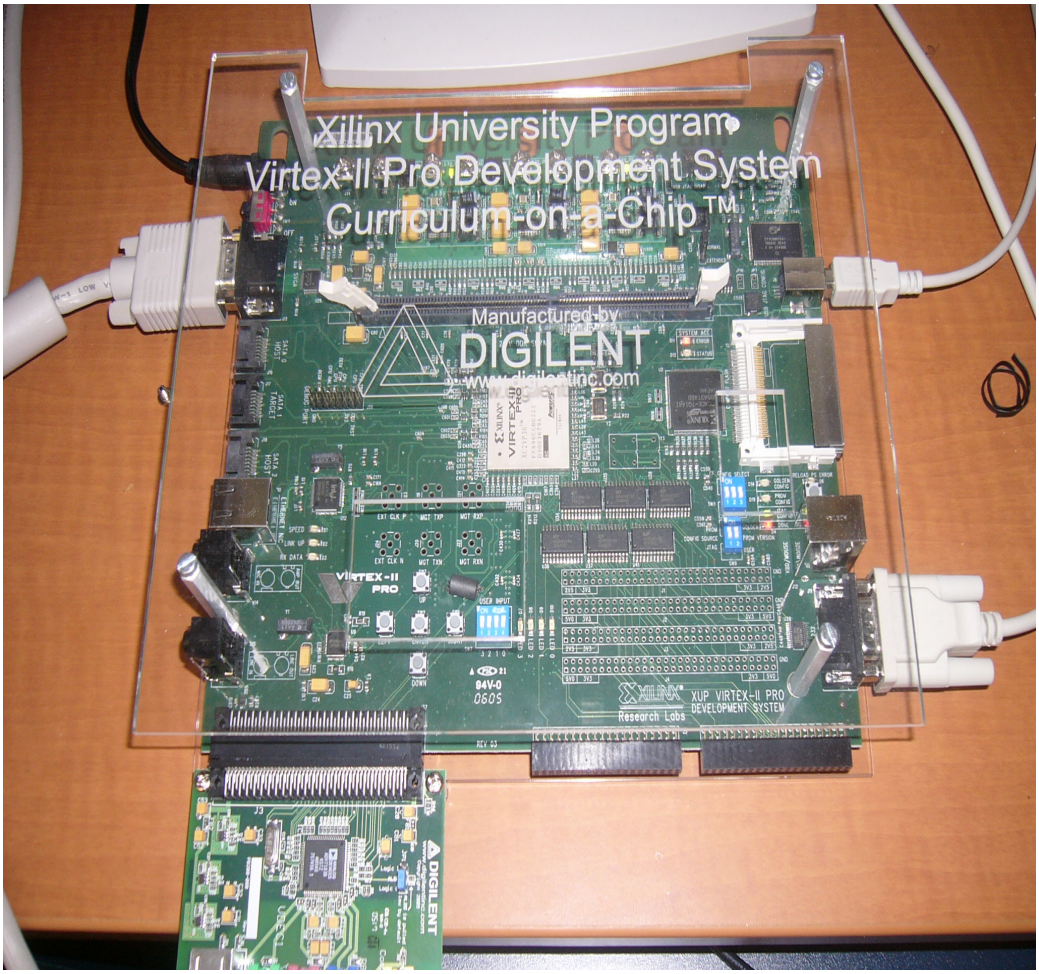
[41] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In Pro-ceedings of Conference on Design Automation and Test in Europe, pages 485–490, 1999.

[42] W. Qin, S. Rajagopalan, and S.Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In Proceedings of the ACM SIGPLAN/SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), June 2004.

[43] Cascade de Criticalblue [En ligne], disponible : http://www.criticalblue.com/criticalblue_products/cascade.shtml

ANNEXES

Annexe 1 : Vue du système de développement de DIGILENT (FPGA Virtex 2 pro)



Annexe 2 : Code VHDL de l'implémentation matérielle

```
----------------------------------------------------------------------------------
-- ECOLE POLYTECHNIQUE DE MONTREAL
-- Engineer: Bouyela Gerard
-- -- Create Date:    13:14:21 05/30/2006
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.VComponents.all;
entity ela_topa is
  Port (        reset           : in  STD_LOGIC;
                clk_s           : in  STD_LOGIC;        --- Horloge du système
                Ho_444_ela  : in  STD_LOGIC;        ----Signal de synchronisation  provenant du module Génération de signaux de
                                                    ---synchronisation et d'horloge de la figure 2-1
                write_address     : in STD_LOGIC_vector(10 downto 0);           --- provenant du controleur d'entree
                pixel_count       : in STD_LOGIC_vector(10 downto 0);           --- provenant du controleur d'entree
                pixel_data_in     : in  STD_LOGIC_vector(23 downto 0);          --- Pixels reçus en entrée
                rgb_out           : out  STD_LOGIC_vector(23 downto 0);         --- Pixels désentrelacés
                clk_27 : in  STD_LOGIC;  --- Pixels désentrelacés
                clk_13 : in  STD_LOGIC); -- Horloge du taux de réception des pixels reçus
end ela_topa;

architecture Behavioral of ela_topa is
component ram_lign is                   ---- mémoire RAM pouvant contenir une ligne de pixels
 Port (         addr : in STD_LOGIC_vector(10 downto 0);
                clk  : in  STD_LOGIC;
                din  : in  STD_LOGIC_vector(23 downto 0);
                dout : out STD_LOGIC_vector(23 downto 0);
                we   : in  STD_LOGIC);
end component;
component BUFG                          --- Buffer pour le clock : évite des 'glichs'
port (          O : out STD_ULOGIC;
                I : in STD_ULOGIC);
end component;
--------------------------- Signaux permettant de contrôler la RAM0 (EN ENTREE)  ---------------------------
 signal addr_ram0  : STD_LOGIC_vector(10 downto 0); signal data_ram_in0  : STD_LOGIC_vector(23 downto 0); signal data_ram_out0 :
STD_LOGIC_vector(23 downto 0);
signal ram_rw_tmp0      : STD_LOGIC;
--------------------------- Signaux permettant de contrôler la RAM1 (EN ENTREE)  ---------------------------
signal addr_ram1 : STD_LOGIC_vector(10 downto 0);signal data_ram_in1  : STD_LOGIC_vector(23 downto 0); signal data_ram_out1 :
STD_LOGIC_vector(23 downto 0):=(others=>'0'); signal ram_rw_tmp1: STD_LOGIC;
--------------------------- Signaux permettant de contrôler la RAM2 (EN ENTREE)  ---------------------------
signal addr_ram2 : STD_LOGIC_vector(10 downto 0);signal data_ram_in2  : STD_LOGIC_vector(23 downto 0);
signal data_ram_out2 : STD_LOGIC_vector(23 downto 0);signal ram_rw_tmp2              : STD_LOGIC;
--------------------------- Signaux permettant de contrôler la RAM0a (EN SORTIE)  ---------------------------
signal addr_ram0a : STD_LOGIC_vector(10 downto 0); signal data_ram_in0a  : STD_LOGIC_vector(23 downto 0);
signal data_ram_out0a : STD_LOGIC_vector(23 downto 0); signal ram_rw_tmp0a          : STD_LOGIC;
--------------------------- Signaux permettant de contrôler la RAM1a (EN SORTIE)  ---------------------------
signal addr_ram1a : STD_LOGIC_vector(10 downto 0); signal data_ram_in1a  : STD_LOGIC_vector(23 downto 0);
signal data_ram_out1a : STD_LOGIC_vector(23 downto 0); signal ram_rw_tmp1a          : STD_LOGIC;
--------------------------- Signaux permettant de contrôler la RAM2a (EN SORTIE)  ---------------------------
signal addr_ram2a                       : STD_LOGIC_vector(10 downto 0);signal data_ram_in2a  : STD_LOGIC_vector(23 downto 0);
signal data_ram_out2a : STD_LOGIC_vector(23 downto 0);signal ram_rw_tmp2a          : STD_LOGIC;
--------------------------- Signaux permettant de contrôler la RAM3a (EN SORTIE)  ---------------------------
signal addr_ram3a : STD_LOGIC_vector(10 downto 0);signal data_ram_in3a  : STD_LOGIC_vector(23 downto 0);
signal data_ram_out3a : STD_LOGIC_vector(23 downto 0);signal ram_rw_tmp3a          : STD_LOGIC;
```

```
---------------------------- Signaux permettant des calculs ----------------------------
constant pix_ligne_max  : integer :=850 ; signal op_cal_count : STD_LOGIC_vector(3 downto 0);
signal data_calc_1 : STD_LOGIC_vector(23 downto 0); signal data_calc_2  : STD_LOGIC_vector(23 downto 0);
signal data_in1: STD_LOGIC_vector(23 downto 0);  signal data_in2 : STD_LOGIC_vector(23 downto 0);
signal data_cacula1  : STD_LOGIC_vector(23 downto 0);signal data_cacula2  : STD_LOGIC_vector(23 downto 0);
signal addr_calcula1 : STD_LOGIC_vector(10 downto 0);signal addr_calcula2 : STD_LOGIC_vector(10 downto 0);
signal addr_calcul1  : STD_LOGIC_vector(10 downto 0);signal addr_calcul2  : STD_LOGIC_vector(10 downto 0);
signal data_in_h1 : STD_LOGIC_vector(23 downto 0); signal data_in_h2 : STD_LOGIC_vector(23 downto 0);
signal data_in_h3 : STD_LOGIC_vector(23 downto 0); signal data_in_b1 : STD_LOGIC_vector(23 downto 0);
signal data_in_b2 : STD_LOGIC_vector(23 downto 0);signal data_in_b3 : STD_LOGIC_vector(23 downto 0);
signal addr_Lhaut: STD_LOGIC_vector(10 downto 0);signal addr_Lbas: STD_LOGIC_vector(10 downto 0);  signal addr_calcula1_t:
STD_LOGIC_vector(10 downto 0);signal addr_calcula2_t: STD_LOGIC_vector(10 downto 0);
signal H_sync : STD_LOGIC;signal op_aff :  STD_LOGIC;signal op_aff_tmp:  STD_LOGIC; signal data_calcul:  STD_LOGIC_vector(23
downto 0);signal op_count : STD_LOGIC_vector(1 downto 0); signal addr:  STD_LOGIC_vector(10 downto 0);signal sync:
                STD_LOGIC_vector(1 downto 0);signal ram_rw_1 :std_logic; ---<='1'; signal clk_rw0 :  STD_LOGIC;signal clk_rw1 :
STD_LOGIC;signal clk_rw2 :  STD_LOGIC;  signal clk_ra0 :  STD_LOGIC;signal clk_ra1 :  STD_LOGIC;signal clk_ra2 :  STD_LOGIC;signal
clk_ra3 :  STD_LOGIC; signal clk_27_ram :  std_logic;signal clk_13_ram :  std_logic; signal clk_s_buf                :  STD_LOGIC; signal
temp_rst:   STD_LOGIC;
begin
------ PORT MAP  POUR LES RAMS D'ENTREE
ram_write0 : ram_lign port map (  addr =>addr_ram0 ,clk =>clk_rw0,din =>data_ram_in0,dout=>data_ram_out0,   we=>ram_rw_tmp0 );
ram_write1 : ram_lign port map ( addr =>addr_ram1 ,clk =>clk_rw1,---,clk_rw1,din =>data_ram_in1,dout=>data_ram_out1
we=>ram_rw_tmp1);
ram_write2 : ram_lign port map ( addr =>addr_ram2 ,clk =>clk_rw2,din =>data_ram_in2,dout=>data_ram_out2, we=>ram_rw_tmp2);
------ PORT MAP  POUR LES RAMS DE SORTIE
 ram_aff0 : ram_lign port map ( addr =>addr_ram0a ,clk =>clk_ra0,---clk_s_buf,din
=>data_ram_in0a,dout=>data_ram_out0a,we=>ram_rw_tmp0a );ram_aff1 : ram_lign port map ( addr =>addr_ram1a ,clk =>clk_ra1,---
clk_s_buf,din =>data_ram_in1a,dout=>data_ram_out1a,we=>ram_rw_tmp1a);ram_aff2 : ram_lign port map ( addr =>addr_ram2a ,clk
=>clk_ra2,---cl_s_buf,din =>data_ram_in2a,dout=>data_ram_out2a,we=>ram_rw_tmp2a);ram_aff3 : ram_lign port map ( addr =>addr_ram3a
,clk =>clk_ra3,---clk_s_buf,din =>data_ram_in3a,dout=>data_ram_out3a,we=>ram_rw_tmp3a);clk_13_ram<=clk_13;clk_27_ram<=clk_27;
-------------------------------------------------------------
---- gestion des sorties des ram d'affichage ; permet de synchroniser la fin de l'écriture d'une ligne
process(reset,clk_13,write_address)
begin
if (reset='1') then
                 op_count<= (others => '0');
                 op_aff<='0';
                 temp_rst<='0';
elsif (clk_13='1' and clk_13'event ) then                ----au front descendant ..... dans le fichier .v au front montant
                if (temp_rst='0') and (write_address=1) then  op_aff<=not op_aff ; temp_rst<='1';
                end if ;
                if (write_address=pix_ligne_max-1) then                 --- quand on debut une ligne --- pix_ligne_max-1 avant
                          if (op_count="10") then   op_count<= (others => '0');
                          else  op_count<=op_count+1 ;
                          end if ;
                          op_aff<=not op_aff ;
                end if ;
end if ;
end process ;
-------------------------------------------------------------
process(reset,clk_27,pixel_count)
begin
if (reset='1') then
                sync<= "10";
elsif (clk_27='1' and clk_27'event ) then                ----au front descendant ..... dans le fichier .v au front montant
                if (pixel_count=pix_ligne_max-1) then                   --- quand on debut une ligne --- pix_ligne_max-1 avant
                           sync<= sync+1;
                end if;
end if;
end process ;
----- gestion de la sortie RGB  controleur de SORTIE
process(sync,reset,clk_27 )
begin
if (reset='1') then  rgb_out<= (others =>'0');
elsif (clk_27='1' and clk_27'event )  then
                  case sync is
                  when "00"=>  rgb_out<= data_ram_out0a;    when "01"=>  rgb_out<= data_ram_out1a;
                  when "10" => rgb_out<= data_ram_out2a;    when "11"=>  rgb_out<= data_ram_out3a;
                  when others =>   rgb_out<= (others =>'0');    end case ;
end if ;
```

```
---- processus permettant de lire les données d'entrées et de remplir les ram se sorties c'est la partie implementant l'algortihme ELA en VHDL
process(clk_s_buf,reset)---,op_aff,data_calc_1,data_calc_2)
variable tmp_calcul1_1: STD_LOGIC_vector (8 downto 0):=(others =>'0'); variable tmp_calcul2_1: STD_LOGIC_vector (8 downto
0):=(others =>'0');
variable tmp_calculr_1: STD_LOGIC_vector (8 downto 0):=(others =>'0'); variable tmp_calcul1_2: STD_LOGIC_vector (8 downto
0):=(others =>'0');
variable tmp_calcul2_2: STD_LOGIC_vector (8 downto 0):=(others =>'0'); variable tmp_calculr_2: STD_LOGIC_vector (8 downto
0):=(others =>'0');
variable tmp_calcul1_3: STD_LOGIC_vector (8 downto 0):=(others =>'0');variable tmp_calcul2_3: STD_LOGIC_vector (8 downto
0):=(others =>'0');
variable tmp_calculr_3: STD_LOGIC_vector (8 downto 0):=(others =>'0');

---- Variable utilisée pour le calcul des pixels ; a la fin du nom de la variable la lettre correspond a la couleur et le chiffre au numéro du pixel
variable Cal_r1: signed (8 downto 0):=(others =>'0');variable Cal_r2: signed (8 downto 0):=(others =>'0');variable Cal_r3: signed (8 downto
0):=(others =>'0');
variable Cal_g1: signed (8 downto 0):=(others =>'0');variable Cal_g2: signed (8 downto 0):=(others =>'0');variable Cal_g3: signed (8 downto
0):=(others =>'0');
variable Cal_b1: signed (8 downto 0):=(others =>'0');variable Cal_b2: signed (8 downto 0):=(others =>'0'); variable Cal_b3: signed (8 downto
0):=(others =>'0');
 begin
 if reset='1' then
                op_cal_count<=(others=>'0') ; data_in1<=(others=>'0') ; data_in2<=(others=>'0') ; data_cacula1<=(others=>'0');
                data_cacula2<=(others=>'0') ; data_calcul<=(others=>'0') ;   addr_calcula1<= (others => '0'); addr_calcula2<= (others => '0');
                addr_calcul1<= (others => '0');
                addr_calcul2<= (others => '0'); addr_Lhaut<= (others => '0'); addr_Lbas<= (others => '0');ram_rw_1<='0';addr<=(others=>'0');

                 data_in_h1 <= (others => '0'); data_in_h2<= (others => '0'); data_in_h3 <= (others => '0'); data_in_b1 <= (others => '0');
                data_in_b2 <= (others => '0'); data_in_b3 <= (others => '0');
elsif (clk_s_buf='1' and clk_s_buf'event) then
                addr_calcula2<=addr;
                addr_calcul1<=addr_Lhaut; --- addresse lign du haut
                addr_calcul2<=addr_Lbas;-- addresse lign du bas
                 case op_cal_count is
                                when "0000" =>
                                                op_aff_tmp<= op_aff; op_cal_count<=op_cal_count+1;
                                                ram_rw_1<='0'; addr_Lhaut<=(others=>'0');addr_Lbas<=(others=>'0');
                                                addr<="00000000001"; data_in_h1 <= (others => '0');  data_in_h2<= (others => '0');
                                                data_in_h3 <= (others => '0'); data_in_b1 <= (others => '0');data_in_b2 <= (others => '0');
                                                data_in_b3 <= (others => '0');
                                when "0001" =>
                                                if (op_aff_tmp=op_aff) then
                                                                op_cal_count<="0001";
                                                else
                                                                op_cal_count<=op_cal_count+1;
                                                                data_in_h1<=data_calc_1;
                                                                data_in_b1<=data_calc_2;
                                                                addr_Lhaut<=addr_Lhaut+1;
                                                                addr_Lbas<=addr_Lbas+1;
                                                end if;
                                when "0010" =>
                                                op_cal_count<=op_cal_count+1;
                                when "0011" =>
                                                 data_in_h2<=data_calc_1;
                                                 data_in_b2<=data_calc_2;
                                                 addr_Lhaut<=addr_Lhaut+1;
                                                 addr_Lbas<=addr_Lbas+1;
                                                 op_cal_count<=op_cal_count+1;
                                when "0100" =>
                                                op_cal_count<=op_cal_count+1;
                                when "0101"=>
                                                data_in_h3<=data_calc_1;
                                                data_in_b3<=data_calc_2;
                                                 addr_Lhaut<=addr_Lhaut+1; ---- pour le prochain
                                                addr_Lbas<=addr_Lbas+1;   ---- pour le prochain
                                                 op_cal_count<=op_cal_count+1;
---- traitement pour R       when "0110"=>
                                                Cal_r1:=  signed(data_in_h1(7 downto 0))- signed(data_in_b3(7 downto 0));
                                                Cal_r2:=  signed(data_in_h2(7 downto 0))- signed(data_in_b2(7 downto 0));
                                                Cal_r3:=  signed(data_in_h3(7 downto 0))- signed(data_in_b1(7 downto 0));
                                                if (Cal_r1<Cal_r2) then
                                                                if ((Cal_r2<Cal_r3) or (Cal_r1< Cal_r3 )) then ---- d'ou Cal_r1 est le plus petit
                                                                                data_in1(7 downto 0)<=data_in_h1(7 downto 0) ;
                                                                                data_in2(7 downto 0)<=data_in_b3(7 downto 0) ;
```

```
---- traitement pour G
                        Cal_g1:= signed(data_in_h1(15 downto 8))- signed(data_in_b3(15 downto 8));
                        Cal_g2:= signed(data_in_h2(15 downto 8))- signed(data_in_b2(15 downto 8));
                        Cal_g3:= signed(data_in_h3(15 downto 8))- signed(data_in_b1(15 downto 8));
                        if (Cal_g1<Cal_g2) then
                                if ((Cal_g2<Cal_g3) or (Cal_g1< Cal_g3 )) then ---- d'ou Cal_r1 est le plus petit
                                        data_in1(15 downto 8)<=data_in_h1(15 downto 8) ;
                                        data_in2(15 downto 8)<=data_in_b3(15 downto 8) ;
                                        else                    --- d'ou Cal_r3 est le plus petit
                                        data_in1(15 downto 8)<=data_in_h3(15 downto 8) ;
                                        data_in2(15 downto 8)<=data_in_b1(15 downto 8) ;end if ;else
                                if ((Cal_g1<Cal_g3) or (Cal_g2<Cal_g3)) then  ---D'ou Cal_r2 est le plus petit
                                        data_in1(15 downto 8)<=data_in_h2(15 downto 8) ;
                                        data_in2(15 downto 8)<=data_in_b2(15 downto 8) ;
                                        else                    --- d'ou Cal_r3 est le plus petit
                                        data_in1(15 downto 8)<=data_in_h3(15 downto 8) ;
                                        data_in2(15 downto 8)<=data_in_b1(15 downto 8) ;end if ;end if ;
---- traitement pour B
                        Cal_b1:= signed(data_in_h1(23 downto  16))- signed(data_in_b3(23 downto 16));
                        Cal_b2:= signed(data_in_h2(23 downto  16))- signed(data_in_b2(23  downto 16));
                        Cal_b3:= signed(data_in_h3(23 downto  16))- signed(data_in_b1(23  downto 16));
                        if (Cal_b1<Cal_b2) then
                                if ((Cal_b2<Cal_b3) or (Cal_b1< Cal_b3 )) then ---- d'ou Cal_r1 est le plus petit
                                        data_in1(23 downto  16)<=data_in_h1(23 downto 16) ;
                                        data_in2(23 downto  16)<=data_in_b3(23 downto 16) ;
                                else                    --- d'ou Cal_r3 est le plus petit
                                        data_in1(23 downto  16)<=data_in_h3(23 downto 16) ;
                                        data_in2(23 downto  16)<=data_in_b1(23 downto 16) ;end if ;
                        else
                                if ((Cal_r1<Cal_r3) or (Cal_r2<Cal_r3)) then  ---D'ou Cal_r2 est le plus petit
                                        data_in1(23 downto 16)<=data_in_h2(23 downto 16) ;
                                        data_in2(23 downto 16)<=data_in_b2(23 downto 16) ;
                                else                    --- d'ou Cal_r3 est le plus petit
                                        data_in1(23 downto 16)<=data_in_h3(23 downto 16) ;
                                        data_in2(23 downto 16)<=data_in_b1(23 downto 16) ;end if ;end if ;
                        op_cal_count<=op_cal_count+1;
                when "0111"=>
                        tmp_calcul1_1 := "000000000"; tmp_calcul2_1 := "000000000";tmp_calculr_1 := "000000000";
                        tmp_calcul1_1(7 downto 0) := data_in1(7 downto 0); tmp_calcul2_1(7 downto 0) := data_in2(7 downto 0);
                        tmp_calculr_1 :=tmp_calcul1_1+tmp_calcul2_1; tmp_calcul1_2 := "000000000"; tmp_calcul2_2 := "000000000";
                        tmp_calculr_2 := "000000000";
                        tmp_calcul1_2(7 downto 0) := data_in1(15 downto 8);
                        tmp_calcul2_2(7 downto 0) := data_in2(15 downto 8);
                        tmp_calculr_2 :=tmp_calcul1_2+tmp_calcul2_2;
                        tmp_calcul1_3 := "000000000";tmp_calcul2_3 := "000000000";tmp_calculr_3 := "000000000";
                        tmp_calcul1_3(7 downto 0) := data_in1(23 downto 16);  tmp_calcul2_3(7 downto 0) := data_in2(23 downto 16);
                        tmp_calculr_3 :=tmp_calcul1_3+tmp_calcul2_3; data_calcul(7 downto 0)   <=tmp_calculr_1(8 downto 1);
                        data_calcul(15 downto 8)  <=tmp_calculr_2(8 downto 1);
                        data_calcul(23 downto 16) <=tmp_calculr_3(8 downto 1);
                        op_cal_count<=op_cal_count+1;
                when "1000"=>
                        data_cacula1 <=data_in_h2;data_cacula2 <=data_calcul;
                        data_in_h1 <= data_in_h2;  ---- on decalle la fenetre
                        data_in_h2 <= data_in_h3;  data_in_b1 <= data_in_b2;  ---- on decalle la fenetre
                        data_in_b2 <= data_in_b3;
                        op_cal_count<=op_cal_count+1;
                when "1001"=>
                        if (addr=pix_ligne_max-1) then ------1
                                addr<=(others=>'0');  op_cal_count<="0000";
                                ram_rw_1<='0'; elseop_cal_count<="0101";
                                addr<=addr+1 ;ram_rw_1<='1'; end if ;
                        if (addr_calcula1=pix_ligne_max-1) then
                                addr_calcula1<="00000000000";---(others=>'0');
                        else            addr_calcula1<=addr_calcula1+1;end if ;

                when others =>
                        addr<=(others=>'0') ;  op_cal_count<=(others=>'0') ;data_in1<=(others=>'0') ;data_in2<=(others=>'0') ;
                        data_cacula1<=(others=>'0') ;data_cacula2<=(others=>'0') ;data_calcul<=(others=>'0') ;addr_calcula1<= (others =>
                        '0');
                        addr_calcula2<= (others => '0');addr_calcul1<= (others => '0');addr_calcul2<= (others => '0');ram_rw_1<='0';
                end case ;
end if;
end process ;
```

```
--- permet de gerer le remplissage des ram d'affichage
process(reset,op_aff,addr_calcula1,addr_calcula2,clk_s_buf)
begin
		if (reset='1') then
			addr_ram0a   <= (others=>'0') ; data_ram_in0a <= (others=>'0') ; ram_rw_tmp0a  <= '0'; clk_ra0<= clk_s_buf;
			addr_ram1a   <= (others=>'0') ; data_ram_in1a <=(others=>'0') ; ram_rw_tmp1a  <= '0'; clk_ra1<= clk_s_buf;
			addr_ram2a   <= (others=>'0') ; data_ram_in2a <=(others=>'0') ; ram_rw_tmp2a  <= '0'; clk_ra2<= clk_s_buf;
			addr_ram3a   <= (others=>'0') ;data_ram_in3a <=(others=>'0') ; ram_rw_tmp3a  <= '0';clk_ra3<= clk_s_buf;
		else
			if (op_aff='1') then
				addr_ram0a    <= addr_calcula1 ;  data_ram_in0a <= data_cacula1  ; ram_rw_tmp0a  <= ram_rw_1;
				clk_ra0<= clk_s_buf;

				addr_ram1a   <= addr_calcula2 ; data_ram_in1a <= data_cacula2  ;ram_rw_tmp1a  <= ram_rw_1;
				clk_ra1<= clk_s_buf;

				addr_ram2a   <= pixel_count  ; ram_rw_tmp2a  <= '0'; clk_ra2<=clk_27_ram;

				addr_ram3a   <= pixel_count ; ram_rw_tmp3a  <= '0'; clk_ra3<=clk_27_ram;
			else
				addr_ram0a   <= pixel_count  ; ram_rw_tmp0a  <= '0'; clk_ra0<=clk_27_ram; .
				addr_ram1a   <= pixel_count  ;ram_rw_tmp1a  <= '0';clk_ra1<=clk_27_ram;
				addr_ram2a    <= addr_calcula1 ;data_ram_in2a <= data_cacula1  ;ram_rw_tmp2a  <= ram_rw_1
				clk_ra2<=clk_s_buf;
				addr_ram3a   <= addr_calcula2 ;data_ram_in3a <= data_cacula2  ;ram_rw_tmp3a  <= ram_rw_1 ;
				clk_ra3<=clk_s_buf;
			end if;
		end if;
end process ;  ---- permet de remplir les 3 rams d'entrée
process(reset,op_count,clk_s_buf)
begin
		if (reset='1') then
			addr_ram0    <= (others => '0'); data_ram_in0 <= (others => '0'); ram_rw_tmp0  <= '0'; clk_rw0 <= clk_s_buf;
			addr_ram1    <=(others => '0'); data_ram_in1<= (others => '0'); ram_rw_tmp1  <= '0';clk_rw1 <= clk_s_buf;
			addr_ram2 <= (others => '0'); data_ram_in2 <= (others => '0'); ram_rw_tmp2  <= '0'; clk_rw2 <= clk_s_buf;
			data_calc_1  <= (others => '0');data_calc_2<=(others => '0');
		else
			case op_count is

				when "00"=>
					addr_ram0    <= write_address ; data_ram_in0 <= pixel_data_in  ; ram_rw_tmp0  <= '1';
					clk_rw0<=clk_13_ram;
					addr_ram1    <= addr_calcul1  ; data_calc_1<=  data_ram_out1  ;ram_rw_tmp1  <= '0';
					clk_rw1<=clk_s_buf;
					addr_ram2 <= addr_calcul2  ; data_calc_2 <= data_ram_out2  ;ram_rw_tmp2  <= '0';
					clk_rw2<=clk_s_buf;
				when "01"=>
					addr_ram0    <= addr_calcul2  ;data_calc_2  <=data_ram_out0;ram_rw_tmp0   <= '0';
					clk_rw0<=clk_s_buf;
					addr_ram1    <=  write_address ;data_ram_in1 <= pixel_data_in; ram_rw_tmp1  <= '1';
					clk_rw1<=clk_13_ram;
					addr_ram2 <= addr_calcul1  ; data_calc_1 <=data_ram_out2   ;ram_rw_tmp2  <= '0';
					clk_rw2<=clk_s_buf;
				when "10"=>
					addr_ram0    <= addr_calcul1  ; data_calc_1   <=data_ram_out0   ;ram_rw_tmp0   <=
					'0';
					clk_rw0<=clk_s_buf;
					addr_ram1    <=  addr_calcul2  ;data_calc_2<=  data_ram_out1  ;ram_rw_tmp1  <= '0';
					clk_rw1<=clk_s_buf;
					addr_ram2 <=  write_address;data_ram_in2<=pixel_data_in   ;ram_rw_tmp2  <= '1';
					clk_rw2<=clk_13_ram;
				when others=>
					addr_ram0    <= (others => '0');data_ram_in0 <= (others => '0');ram_rw_tmp0  <= '0';
					clk_rw0<=clk_s_buf;
					addr_ram1    <=(others => '0');data_calc_1<= (others => '0');ram_rw_tmp1  <= '0';
					clk_rw1<=clk_s_buf;
					addr_ram2 <= (others => '0');data_calc_2<=(others => '0');ram_rw_tmp2  <= '0';
					clk_rw2<=clk_s_buf;
				end case ;
		end if; end process ;
end Behavioral;
```

Annexe 3 : Exemple de code Lisa main.LISA (version R3)

```
/*
Code main.lisa de l<architecture R3

*/
#include "defines.h"
#include "memory_cfg.h"
#include "memory_if.h"
/* Specify a version string which is contained later
   in the generated tools */
VERSION("LT_RISC_32p5, 2005.2.0")

 /*================================================*/
 /* Description des ressourses du processeur       */
 /*================================================*/

RESOURCE
{
 /* Register file with 16 registers      */
 REGISTER uint32 R[0..15];
 /* Fetch program counter register*/
 REGISTER        uint32 FPC;
 /* Program counter register*/
 PROGRAM_COUNTER uint32 PC;
 /* Stack pointer for profiling */
 /* In fact, this is the frame pointer, but we just need one
  * register which changes at a constant offset from the
  * start of a function. The write to the frame pointer is
  * generally the second instruction in each function */
 STACK_POINTER    TClocked<uint32> FP ALIAS R[13];
 /* Status register */
 REGISTER        bool Z, N, C, V;  REGISTER  bool   BSET;  REGISTER        bool   BSET6 ;
 REGISTER        uint16 BPC;   REGISTER  int saut;
 /* 3 stage pipeline */
 PIPELINE   pipe = { FE ; DC ; EX ; MEM; WB };
 /* A pipeline register to pass data through the pipeline */
 PIPELINE_REGISTER IN pipe
   {
    PROGRAM_COUNTER uint16 pc;      /* the address of the instruction */
    uint32 insn;    /* the instruction word */
    uint8  rs1;     /* operand register 1 address */
    uint8  rs2;     /* operand register 2 address */
    uint8  rs3;     /* shifter operand register address */
    uint8  rs4;          uint8  rs5;          uint8  rs6;          uint32 op1;     /* operand 1 value */      uint32 op2;     /* operand 2 value */
    uint32 op3;     /* shifter operand value */      uint32 op4;     /* operand 1 value */
    uint32 op5;     /* operand 2 value */      uint32 op6;     /* shifter operand value */
    uint16 offset;  /* addr offset */      uint16 address; /* address */
    uint8   BPR;     /* bypass register, destination register address */
    uint32  WBV;     /* writeback value */      uint32 op1m;     /* operand 1 value */      uint32 op2m;     /* operand 2 value */
    uint8  rs1m;     /* operand register 1 address */      uint8  rs2m;     /* operand register 2 address */
   };
 UNIT FETCH    { fetch;     };
 UNIT PIPE_MEM { update_pc; };
}
/*================================================*/
/* Processor Instruction Set Description          */
/*================================================*/
OPERATION reset
{
 BEHAVIOR
   {
    /* C Behavior Code to reset all the processor         resources to a well defined state */      int i;
    /* Zero register file */      for (i = 0 ; i < 16 ; i++)
                r
```

```
    CLEAR_Z_FLAG;CLEAR_N_FLAG; CLEAR_C_FLAG;  CLEAR_V_FLAG;
        /* Set program counter to the program entry point*/
    FPC = LISA_PROGRAM_COUNTER;
     /* Clear the entire pipeline */
    PIPELINE(pipe).flush();
    BPC  = 0;     BSET = 0;                BSET6  = 0;     PC   = 0;
    shifter_in1 = 0;   shifter_in1 = 0;    shifter_in2 = 0;     shifter_out = 0;     branch_address = 0;     branch_offset = 0;
    store_op = 0;     alu_in1 = 0;     alu_in2 = 0;     alu_out = 0;
    FP = 0;     V = 0;     C = 0;     N = 0;     Z = 0;
#pragma analyze(off)
    init_syscall();
#pragma analyze(on)
   }
}
OPERATION main
{
 /* Operation main is triggered every control step (clock) */
 DECLARE
   {
    INSTANCE fetch, update_pc;
   }
   BEHAVIOR
   {
    /* Execute all activated operations in the pipeline */
    PIPELINE(pipe).execute();
    /* Advance the pipeline by one cycle */
    PIPELINE(pipe).shift();
   }
 ACTIVATION {
  /* Always update current pc */
  update_pc,
   /* Activate the operation "fetch". This will put "fetch" into the pipeline.
              The pipeline-scheduler will automatically execute it when the time is appropriate */
   if (!pipe.DC.IN.stalled())
              {
               fetch
              }
   }
}
/* The operation fetch is now assigned to the FE stage. This assignment is necessary to
  define the appropriate timing for the execution of this operation */
OPERATION fetch IN pipe.FE
{
 DECLARE
   {
    INSTANCE decode;
   }
 BEHAVIOR
   {
    uint32 tmp;
     if (BSET)
              {
                tmp = BPC;
               BSET = 0;
              }
    else
              {
               tmp = FPC;
              }
    /* Now we put the instruction word and the address into our pipeline register between FE and DC stage */
    OUT.pc = tmp;
     /* Load instruction word from program memory, macro is defined in memory_if.h */
    PMEM_LW(OUT.insn, tmp);
    /* Reset the alu bypass register */
    /* Note: BPR contained the bitwise not of the index of the register that
     * is bypassed (otherwise, a flush would set the bypass to register 0 */
    OUT.BPR = 0;
    /* Increment the fetch program counter to the next fetch address*/
    FPC = tmp + 4;
   }
```

```
  ACTIVATION
    {
     /* Activate the operation "decode". This will put "decode" into the pipeline.
                The pipeline-scheduler will automatically execute it when the time is appropriate */
     decode
    }
}

/* The operation decode is now assigned to the DC stage. This assignment is necessary to
   define the appropriate timing for the execution of this operation */

OPERATION decode IN pipe.DC
{
  DECLARE
    {
      GROUP insn_set = {
minimun || moyenne2 ||  abs_f || elal|


       alu_rri || alu_rrr || alu_rrrr || alu_rrri || lui_ri || ldc_ri  ||
       ld_rr || st_rr || nop || bra || brai || brau || cmp_rr || cmp_ri ||
                trapi
     };
    }
  CODING AT (  IN.pc ) { IN.insn == insn_set  }
  SYNTAX { insn_set }
  BEHAVIOR { /* Nothing to b done */ }
  ACTIVATION { insn_set }
}
/* Update the pc only when MEM stage is not stalled */
OPERATION update_pc IN pipe.MEM
{
  BEHAVIOR {
    if (MEM.IN.pc) {
      /* PC in pipe reg is valid. This PC update is only important for the
       * debugger display and GDB. */
      PC = MEM.IN.pc;
    }
  }
}
```

Annexe 4 : Exemple de code Lisa ELA.LISA (version R3)

```
#include "defines.h"
 ////////////////////////////////// defintion des ressources de nostre fonction
RESOURCE
{
 uint32 rgb_h1;  uint32 rgb_h2;  uint32 rgb_h3;  uint32 rgb_b1;  uint32 rgb_b2;  uint32 rgb_b3;
 uint32 rgb_out;  UNIT ela_DC { ela; };  UNIT ela_EXe { ela_ex; };
}
INSTRUCTION ela IN pipe.DC
{
 DECLARE
   {
/* ELA instructions */ INSTANCE ela_ex;
/* source registers*/ GROUP dst,src1,src2,src3,src4,src5,src6   ={ reg};
INSTANCE  bypass_src1_dc, bypass_src2_dc , bypass_src3_dc,bypass_src4_dc, bypass_src5_dc , bypass_src6_dc;
   }
   CODING { 0b0111 dst src1 src2 src3 src4 src5 src6  }  ////
   SYNTAX {"ela(" dst","src1","src2","src3","src4","src5","src6 ")"}
   BEHAVIOR
   {
    /* Put the source registers into the pipe to allow forwarding to EX */
      bypass_src1_dc();   bypass_src2_dc(); bypass_src3_dc();  bypass_src4_dc(); bypass_src5_dc();  bypass_src6_dc();
   }
    ACTIVATION
   {
    ela_ex
   }
}
 OPERATION ela_ex IN pipe.EX
{
 DECLARE
   {
    REFERENCE  dst;       INSTANCE  writeback_dst;
   }
 BEHAVIOR
   {
    uint32 XR16, XR25,XR34, XG16, XG25, XG34, XB16, XB25, XB34;
    uint32  diffR16,diffR25,diffR34;  uint32  diffG16,diffG25,diffG34;  uint32  diffB16,diffB25,diffB34;
    uint32 G1,G2,G3,G4,G5,G6;
    uint32 B1,B2,B3,B4,B5,B6;.
    uint32 Rtmp,Gtmp,Btmp;
   if (BYPASS_ACTIVE(MEM.IN.BPR,IN.rs1))  { rgb_h1 = MEM.IN.WBV; }
    else  {
                if (BYPASS_ACTIVE(WB.IN.BPR,IN.rs1))   {  rgb_h1 = WB.IN.WBV;   }
                else    {rgb_h1 = IN.op1;  }
          }
  if (BYPASS_ACTIVE(MEM.IN.BPR,IN.rs2)) {  rgb_h2 = MEM.IN.WBV; }
    else   {
                if (BYPASS_ACTIVE(WB.IN.BPR,IN.rs2))  { rgb_h2 = WB.IN.WBV;         }
                else  { rgb_h2 = IN.op2;}
           }
   if (BYPASS_ACTIVE(MEM.IN.BPR,IN.rs3))  { rgb_h3 = MEM.IN.WBV;               }
   else{
                if (BYPASS_ACTIVE(WB.IN.BPR,IN.rs3))   {  rgb_h3 = WB.IN.WBV;     }
                else   {     rgb_h3 = IN.op3;   }
      }
    if (BYPASS_ACTIVE(MEM.IN.BPR,IN.rs4)) {  rgb_b1 = MEM.IN.WBV; }
    else {
                if (BYPASS_ACTIVE(WB.IN.BPR,IN.rs4))   {     rgb_b1 = WB.IN.WBV;    }
                else   {   rgb_b1 = IN.op4;   }
          }
    if (BYPASS_ACTIVE(MEM.IN.BPR,IN.rs5)) {  rgb_b2 = MEM.IN.WBV; }
    else  {
                if (BYPASS_ACTIVE(WB.IN.BPR,IN.rs5))   {    rgb_b2 = WB.IN.WBV;   }
                else   {    rgb_b2 = IN.op5; }
          }
```

```
///// Partie decrivant l<algorithme ELA
                B1 = ( rgb_h1 & 0x000000ff );  G1 = ((rgb_h1 >> 8) & 0x000000ff ); rgb_h1 = (( rgb_h1 >> 16) & 0x000000ff );
                B2 = (rgb_h2 & 0x000000ff );  G2 = (( rgb_h2 >> 8) & 0x000000ff ); rgb_h2 = (( rgb_h2 >> 16) & 0x000000ff );
                B3 = ( rgb_h3 & 0x000000ff ); G3 = (( rgb_h3 >> 8) & 0x000000ff ); rgb_h3 = ((rgb_h3 >> 16) & 0x000000ff );
                B4 = (rgb_b1 & 0x000000ff );   G4 = ((rgb_b1 >> 8) & 0x000000ff ); rgb_b1 = (( rgb_b1 >> 16) & 0x000000ff );
                B5 = ( rgb_b2 & 0x000000ff );  G5 = (( rgb_b2 >> 8) & 0x000000ff ); rgb_b2 = ((rgb_b2 >> 16) & 0x000000ff );
                B6 = (rgb_b3 & 0x000000ff ); G6 = ((rgb_b3 >> 8) & 0x000000ff );  rgb_b3 = ((rgb_b3 >> 16) & 0x000000ff );
  /// Traitement de la couleur Rouge
                 /// Calcul de la valeur absolue  des direction d'interpolation
                 diffR16 = rgb_h1 - rgb_b3;   diffR25 = rgb_h2 - rgb_b2;  diffR34 = rgb_h3 - rgb_b1;
               if ((diffR16 >>31) == 0) {XR16 = diffR16 ;} else {XR16 = - diffR16; };
               if ((diffR25 >>31) == 0) {XR25 = diffR25 ;} else {XR25 = - diffR25; };
               if ((diffR34 >>31) == 0) {XR34 = diffR34 ;} else {XR34 = - diffR34 ;};
              /// Selon le maximun de vraisemblance des pixels on calcule les pixels manquant
              if (XR16 < XR25)
                {
                 if (XR16 < XR34)  {   Rtmp = ((rgb_h1 + rgb_b3)>>1)&0x000000ff;  }
                 else  {  Rtmp = ((rgb_h3 + rgb_b1)>>1)&0x000000ff; }
                }
                else
                {
                 if (XR25 < XR34) {   Rtmp =  ((rgb_h2 + rgb_b2)>>1)&0x000000ff; }
                 else  {  Rtmp =  ((rgb_h3 + rgb_b1)>>1)&0x000000ff; }
                }
/// Traitement de la couleur Verte
                diffG16 = G1 -  G6;   diffG25 =  G2 -  G5;              diffG34 =  G3 -  G4;
               if ((diffG16>>31) == 0) {XG16 = diffG16 ;} else {XG16 = - diffG16 ;};
               if ((diffG25 >>31) == 0) {XG25 = diffG25; } else {XG25 = - diffG25 ;};
              if ((diffG34>>31) == 0) {XG34 = diffG34; } else {XG34 = - diffG34 ;};
        if (XG16 < XG25)
                {
                 if (XG16 < XG34) {   Gtmp = ((G1 + G6)>>1)&0x000000ff; }
                          else  {  Gtmp = ( (G3 + G4)>>1)&0x000000ff; }
                }
                else
                {
                 if (XG25 < XG34) {  Gtmp =  ((G2 + G5)>>1)&0x000000ff;                                    }
              else  {   Gtmp = ( (G3 + G4)>>1)&0x000000ff;      }
                }
/// Traitement de la couleur Bleu
                diffB16 =  B1 -  B6;   diffB25 =  B2 -  B5;   diffB34 =  B3 -  B4;
                if ((diffB16>>31) == 0) {XB16 = diffB16; } else {XB16 = - diffB16 ;};
                if ((diffB25 >>31) == 0) {XB25 = diffB25; } else {XB25 = - diffB25; };
                 if ((diffB34 >>31) == 0) {XB34 = diffB34 ;} else {XB34 = - diffB34 ;};
                if (XB16 < XB25)
                {                  if (XB16 < XB34) {   Btmp =  ((B1 + B6)>>1)&0x000000ff; }
                          else  {   Btmp =  ((B3 + B4)>>1)&0x000000ff;       }
                }
                else
                {      if (XB25 < XB34) {   Btmp =  ((B2 + B5)>>1)&0x000000ff;  }
                          else  {   Btmp =  ((B3 + B4)>>1)&0x000000ff;  }
                }
                 rgb_out  = ((Rtmp << 16) | (Gtmp << 8) | Btmp ) & 0x00ffffff;
    if (dst != 0)
                {
                 /* set bypass value */
                 OUT.WBV = rgb_out;
                 /* set bypass register*/
                 OUT.BPR = dst;
                }
}
 ACTIVATION { writeback_dst }

}
```